

9-10 ИНФОРМАТИКА

КЛАС

ЗАДЪЛЖИТЕЛНА ПОДГОТОВКА

издателство
ИЗКУСТВА

- Красимир Манев •
- Павел Петров •
- Велислава Христова •
- Нели Манева •
- Бисерка Йовчева •
- Петър Петров •

София, 2012



Информатика
за задължителна подготовка
в 9.-10. клас

Авторски колектив: **Красимир Манев, Павел Петров, Велислава Христова,
Нели Манева, Бисерка Йовчева, Петър Петров**

Редактор на издателството *Ясена Христова*
Предпечатна подготовка и графичен дизайн *Ясена Христова*

Българска. Първо издание
Формат 60x84/8. Печатни коли 18

Издателство „Изкуства“, 1124 София, ул. „Св. Тертер“ №17,
тел. 02/943 4724, факс 02/943 4397
e-mail: izkustva@yahoo.com, www.izkustva.net

- © Красимир Манев, Павел Петров, Велислава Христова,
Нели Манева, Бисерка Йовчева, Петър Петров, автори, 2012
- © Ясена Валериева Христова, предпечатна подготовка и графичен дизайн, 2012
- © Издателство „Изкуства“, 2012

Съдържание

Въведение	5
I. Въведение в информатиката	7
1. Информация и информатика	7
2. Компютри и програми	10
3. Бройни системи	14
4. Съхраняване на данни в компютъра	18
5. Система за програмиране	21
6. Средата Microsoft Visual C# 2010 Express	24
7. Алгоритми	28
II. Въведение в програмирането	34
8. Кратка история на програмирането	34
9. Обектно и събитийно програмиране	37
10. Програма на C# – основни елементи	40
11. Променливи и константи. Инициализация	43
12-13. Операции и изрази	46
14. Стандартен клас математически методи	50
15. Въвеждане и извеждане на данни	52
16. Форматиране на извежданата информация	55
17. Примери за конзолни приложения	58
18. Разклоняване на програмата	60
19. Програми с условен оператор	63
20. Тест: Променливи. Условен оператор	65
III. Програми с графичен интерфейс	67
21. Изграждане на графичния интерфейс	67
22. Основни компоненти	70
23. Основни компоненти – упражнение	74
24-25. Проект: периметър и лице на фигури	76
26-27. Проект: обработка на грешките	78
28. Оформяне на проекта	80
29. Самостоятелна работа	82
30. Реализация на алгоритъм в метод	83
31. Генериране на случайни числа	86
32-33. Проект: калкулатор	87
IV. Цикли и масиви	90
34. Оператор for	90
35. Оператори while и do...while	95
36-37. Калкулатор за гроби	98
38. Едномерен масив	101
39. Едномерен масив. Упражнение	103
40. Вложени цикли. Сортиране	105
41-42. Двумерен масив	108
43. Масиви. Тест	111
44-45. Проект: Статистика за паралелка	112

Съдържание

V. Низове	117
47. Типовете char и string	117
48-49. Операции с низове	120
50-51. Проект Операции с низове	123
52. Сравняване на низове	126
53. Низове (упражнение)	128
54. Тест Низове.....	129
VI. Графика в C#	131
55. Въведение в графиката на C#	131
56-57. Геометрични фигури.....	133
58. Упражнение	136
59. Правоъгълник, елипса и дъга от елипса	137
60. Запълнение на затворени фигури	139
61. Проект с компютърна графика	141
62. Графика на компютърна функция	141
63-64. Първи опити за анимация	144

Въведение

Настоящото учебно помагало е предназначено за учениците от средните училища, изучаващи предмета Информатика в часовете за задължителна подготовка. Съдържанието е структурирано в 64 урочни единици, повечето от които с обем подходящ за преподаване в един учебен час, за да може да се използва както от ученици, изучаващи предмета в един учебен час седмично в 9. и 10. клас, така и от ученици изучаващи предмета в два учебни часа седмично в 9. клас. Съдържанието покрива до голяма степен действащата, от много години и до момента, учебна програма за задължителна подготовка, но е допълнено с теми, съответстващи на съвременните тенденции в развитието на дисциплината.

Информатиката е науката, която се занимава с автоматичната обработка на информацията. Основен обект за изучаване в информатиката е информацията, като философски феномен, нейното представяне във вид, който може да бъде възприеман от човек и съхраняван на различни носители (данни), както и методите за автоматизирана (програмна) обработка на информацията. Основни инструменти на Информатиката са компютърът (хардуерът) и управляващите работата му програми (софтуерът). Затова учениците завършващи средно образование трябва да познават принципите на работа на съвременните компютри и същността на програмирането.

От уроците по Информационни технологии в 5.–8. клас, учениците познават до голяма степен елементите на типичната съвременна компютърна конфигурация, възможностите на операционните системи с графичен интерфейс и редица приложни програми. Това ни освобождава от необходимостта да повтаряме в подробности всички тези изисквани от програмата, но вече изучени доста подробно теми. Освободеният по този начин ресурс от време е посветен на програмирането, и по-точно на представянето на съвременните програмистки парадигми – обектно-ориентираното, визуалното и събитийното програмиране.

Желанието да представим съвременните тенденции в програмирането пред широк кръг от ученици предопредели до голяма степен езикът за програмиране, който е използван за илюстриране на тези тенденции – C# (четем „си шарп“) и средата Visual Studio на Microsoft. Езикът C# е език за обектно-ориентирано програмиране, произлязъл от един от най-добрите езици за професионално програмиране – езикът C. Лишен от претенциозността и тежките конструкции на друг наследник на C – езикът C++, той е отлично средство за запознаване с процеса програмиране на хора, които не се готвят да стават професионални програмисти. В учебното помагало е използвана версията на езика от средата за визуално и събитийно програмиране Visual Studio 2010 Express, но издателството и авторският колектив са готови да окажат необходимото съдействие на всички, които по различни причини използват друга версия на средата за програмиране.

В някои уроци, част от текста е поставен в цветни карета. Така са оформени дефиниции на необходими за изложението понятия, важни концепции или формални описания на конструкциите на езика за програмиране C#. Читателят трябва да обърне сериозно внимание на тези текстове, да се постарее да разбере написаното и да го запомни.

Четенето на учебното помагало в никакъв случай не е достатъчно за овладяване на включения в учебника материал. В голямата част от уроците се разглеждат програмни концепции,

които са илюстрирани със съответни етюди – програми или програмни фрагменти. Добре е тези програми и/или фрагменти да се въвеждат в средата, да се добавя програмен код там, където е необходимо и да се компилират до работещи решения. Практически всеки от уроците включва и специализиран раздел ■ **Работа с компютър**, в който програмно се решава някаква реалистична задача. Този раздел съдържа необходимите разяснения и, обикновено, завършва с пълния код на програмата, ако това е необходимо. Тези програми задължително трябва да се въвеждат в средата, да се компилират и тестват. Някои от уроците са посветени изцяло на работата с компютър и в такъв случай пред заглавието на урока е поставен знакът ■. Повечето от уроците завършват с раздел **Въпроси и задачи**, в който преобладават задачите за самостоятелна работа, изискващи написване на програма. Най-често такава програма може да се получи с неголеми модификации на изучаван в клас „образец“. Целта на помагалото като цяло е, в края на обучението всеки ученик да е в състояние да модифицира изучавани „образци“ до програма с буквено-цифров или графичен интерфейс, която решава практическа задача, с несложен алгоритъм. За улеснение на потребителите, изходните текстове на програмите от учебника, допълнителни упражнения и разработки можете да намерите на сайта www.it.izkustva.net.

Всички програми и елементи от програми в уроците са форматиращи с шрифта `Consolas`, за да подсказват на читателя за предназначението си. По същите причини, всички имена на програми, имена във файловата система, имена на прозорци и графични компоненти, както и свойствата на графичните компоненти са форматиращи с шрифта `Arial`. Когато показваме синтаксисът на някой оператор на езика `C#` или извикване на метод, тогава всички неменящи се части са форматиращи с шрифта `Consolas`, аменящите се – с шрифта `Times New Roman` и са поставени между знаците `<` и `>`.

Голяма част от английските думи използвани в Информатиката вече познаваме от уроците по Информационни технологии (ИТ). В края на всеки раздел е добавен неголям **Речник** със срещащите се в раздела непознати английски думи.

Надяваме се, че предлаганото учебно помагало ще позволи на българските ученици да се запознаят с основите на интересна и ползотворна област на знанието и да придобият практически умения, които ще им бъдат полезни в живота, независимо с какво се занимават. А за тези от тях, които решат да направят програмирането своя професия, подготвяме съответни помагала за часовете в Задължително избираема, Свободно избираема и Професионална подготовка.

Въведение в информатиката

1 Информация и информатика

Науката информатика

От уроците по Информационни технологии познаваме огромните възможности на съвременната компютърна техника да решава различни задачи, които ежедневието поставя, да обогатява познанията ни за света и да ни помага да прекарваме по-интересно и по-приятно свободното си време. Всичко това става благодарение както на развитието на техниката, позволило да се създават компютри с все по-малки и по-малки размери, така и на науката, занимаваща се с компютърното програмиране, наречена *информатика*.

Отделни елементи на това, което днес наричаме информатика, се появяват още през 17 век. Истинските ѝ основи са поставени през 30-те и 40-те години на XX век, а обособяването ѝ като самостоятелна наука става преди не повече от 60 години. Тя, обаче, се развива изключително бързо и днес почти няма област в живота ни, която да не използва постиженията на информатиката.

Думата **ИНФОРМАТИКА** е съставена от думите **ИНФОР**мация и авто**МАТИКА**, т.е. това е науката, която изучава възможностите, начините и средствата за автоматична обработка на информацията.

За да разберем същността на тази дефиниция, трябва да изясним понятията *информация* и *автоматична обработка*.

Информация

Думата *информация* произлиза от латинския глагол *informare*, което означава *оформям, придавам форма на нещо*. Смесът на думата, според Речника на българския език, е „съобщение, сведения за някого или за нещо, осведомяване“.

Всяко нещо на този свят – обект, явление, процес и т.н. се отличава с някакви характерни особености. Както обектът, така и характеристиките, може да са *реални* или *абстрактни* (въображаеми). Например, височината и теглото на един човек са реални характеристики, а името му и неговият ЕГН – абстрактни. Точките в геометрията са абстрактни обекти, а техните характеристики – координатите – са също абстрактни. Характеристиките на абстрактните обекти винаги са абстрактни.

Очевидно е, че една характеристика е съществена тогава, когато е различна за поне два обекта, а ако е характерна само за един обект – когато се мени във времето. Ако всички точки на пространството имаха една и съща температура, например, характеристиката „температура“ не би предизвикала интереса ни.

Характеристиките на обект (явление, процес и т.н.), оформени по начин, който да може да бъде възприеман от човека, наричаме *информация* за този обект.

Познаването на характеристиките на обектите е много важно. Всеки биологичен вид се развива в среда с много важни за съществуването му обекти. Ако не притежава способността да извлича информация за обкръжаващия го свят, да я оценява и да реагира в съответствие с направените оценки, съответният вид изчезва. Човешкият вид се отличава с възможността си да **натрупва информация**, да я **предава** от поколение на поколение и да я **използва**, за да се приспособи към промените на средата,

да я изменя в своя полза и да създава нови обекти с нужните му характеристики и с избрани техни стойности.

Човек възприема някои от реалните характеристики на обектите благодарение на своите *сетива* – зрение, слух, осезание, обоняние и вкус. За установяването на други характеристики са необходими специализирани *измервателни инструменти* – измервателна линия, кантар, термометър и т.н. За да могат стойностите на реални характеристики, наблюдавани от един човек, да се предадат на друг човек, е необходимо те да се представят по някакъв разбираем начин, т.е. да се превърнат в информация.

Абстрактните характеристики не могат нито да се възприемат сетивно, нито да се измерят с инструмент. Абстрактните характеристики се създават от човека, „закрепят“ се за съответния обект и стават информация за този обект. Така например, името на човека и неговият ЕГН се вписват в гражданския регистър. А координатите на избрани точки от земната повърхност – в геодезическите регистри.

Съхраняване и обработване на информация

Натрупването на информация през вековете е огромно постижение на човечеството, но същевременно поражда и проблеми. Един от тези проблеми е **съхраняването на информацията**. Първобитните хора са съхранявали наученото от тях под формата на рисунки върху стените на пещерите, в които са живеели (Фиг. 1). С възникването на писмеността, човечеството създадо удобен инструмент за съхраняване на информацията в писмен вид върху различни носители: папирусите в древен Египет, глинените плочки във Вавилон, пергаментите в древна Европа и т.н.

Многобройните и трудни за запомняне знаци – *йероглифи*, обозначаващи различни понятия, постепенно били заменени с малко на брой, нямащи собствено значение знаци – *букви* – така, че всяка информация може да бъде изразена с комбинации от букви. С изобретяването на хартията и книгопечатането (Фиг. 2) човечеството за дълго време е решило въпроса със съхраняване на натрупаната информация и предаването ѝ на бъдещите поколения.

Простото наблюдение и измерване на обективни характеристики, обаче, не е било достатъчно за развитието на човечеството. За да се реши сложен проблем, освен събирането на информацията, от която зависи решаването на проблема, е необходимо тази информация да бъде **обработена** – да се прегледа внимателно, да се определят измежду многобройните характеристики тези, които са най-важни, да се съпоставят стойностите на важните характеристики и да се търсят връзки между тях (както в случая с повишаването на температурата и увеличаване на обема на телата), да се създават абстрактни понятия и характеристики на реалните обекти (както абстрактните координати, например) и да се използват тези характеристики за получаване на нова информация.

Неудобството на книжните носители е в това, че те **не са подходящи за обработване на наличната в тях информация** – в тях **трудно се търси**. Затова книгите се снабдяват със съдържание или различни индекси на понятията, а за търсене в океана от книги (Фиг. 3) са изобретени библиотечните каталози. С непрекъснатото нарастване на обема на информацията, обаче, тези примитивни средства стават недостатъчни.

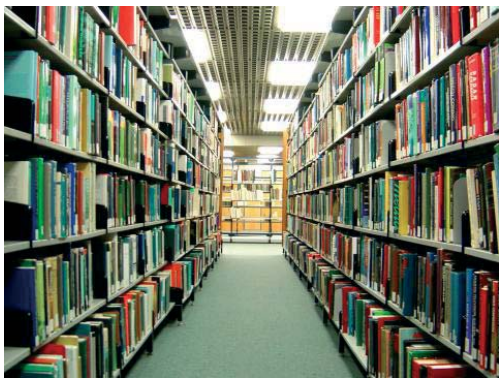
Друг проблем е, че способностите на човек да **съпоставя информация** с цел да извлече важни следствия и да вземе **важни решения** са ограничени. При огромните обеми от натрупана информация един човек и даже големи групи от хора не са в състояние да пре-



Фиг. 1. Пещерни рисунки



Фиг. 2. Печатарска машина

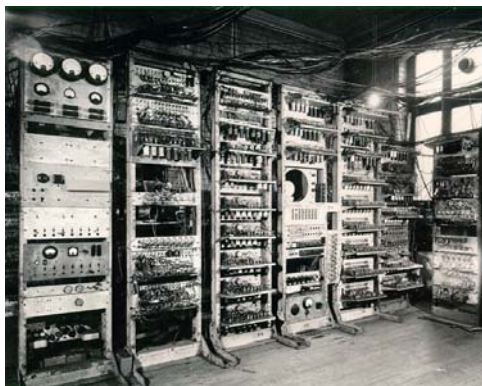


Фиг. 3. Библиотека

гледат всичко изписано по даден проблем и да извлекат нужната за решаването му информация за **разумно време**. В същото време, човечеството непрекъснато изгражда сложни технически системи – ядрените реактори, например, управлението на които изисква обработването на големи обеми информация за много кратко време. Затова, за да не спре в развитието си и усъвършенстването си, човечеството е трябвало да се справи и с този проблем. Натрупаното знание още преди векове позволява той да бъде решен теоретично. От тогава до днес, едни от най-великите умове на човечеството правят стъпка след стъпка за практическото му решаване, но едва през 30-те и 40-те години на миналия век напредъкът на технологиите дава реална възможност това да се случи.

Автоматизирано обработване на информацията

Естествен подход за решаване на проблема е създаването на **машина**, която да усилва интелектуалната мощ на човека, подобно на машините, които той вече е създал, за да увеличи физическата си мощ. За да може да се справи с тежките задачи, които трябва да решава такава машина, тя трябва да е изключително **бърза**. Тук даже повторемостта на работния цикъл на съвременните автомобили от няколко хиляди пъти в минута е абсолютно недостатъчна. В този смисъл е било изключено при работата на машината да се налага намеса от страна на човека, както често става с останалите машини. Такъв начин на работа на една машина, при който намесата на човек е ненужна, наричаме *автоматичен*. Към средата на 40-те години на миналия век започва построяването на първите машини за автоматична обработка на информация – *компютрите* (Фиг. 4). С построяването на първите компютри възниква и науката *компютърна информатика*, а информацията, представена така, че да може да се съхранява в компютрите, започваме да наричаме *данни*.



Фиг. 4. Компютърът MARK I

Създаването на машина, която може автоматично да обработва данни, промени принципно възможностите на човека да изучава и променя обкръжаващата го действителност. Сложните задачи на математическата физика за пръв път получиха шанс да бъдат задоволително решени. Ядрената енергетика, космическите изследвания, моделирането на икономическите процеси, телекомуникациите и ред други области осъществиха огромен напредък, благодарение на използването на компютърна техника.

Умните машини са приложими не само за решаване на сложни научни задачи. В настоящия момент трудно може да се намери област на човешката дейност, където да не се използват, по-малко или повече, компютри. Подготовка на печатни материали, управление на произ-


водство, финанси и персонал на различни по големина фирми, инженерно проектиране (включително на нови компютърни системи), публична администрация, банково дело, образование, развлекателни (но и сериозни) игри – това са само част от областите, в които компютрите са навлезли трайно.

Всичко това стана възможно, защото компютрите, освен да обработват данни, са в състояние да съхраняват огромни количества данни. Устройството за съхраняване на данни на нормален домашен персонален компютър днес е в състояние да съхрани съдържанието на около 1 000 000 средно големи книги, а малката, събираща се в джоб, флаш-памет – около 10 000. Ако приемем, че в целия свят в момента работят един милиард компютърни системи, можем да си представим, макар и приблизително, обема на наличната съхранена информация. В сравнение с класическото хранилище на информация от началото на миналия век – библиотеката, има една съществена разлика. Компютърът е в състояние да

предостави съхранените в него данни за броени секунди. Със създаването на световната **компютърна мрежа** Интернет, цялата натрупана до момента информация стана достъпна практически от всяка точка на земното кълбо, до която е достигнала мрежата.

Би могло да се мисли, че за краткия период на своето съществуване информатиката е достигнала своя апогей и практически е изчерпала възможностите си. Вселената, Човекът и неговите възможности да се развива и променя, изглежда нямат край.

Въпроси и задачи

1. Посочете в природата източници на информация, които се възприемат директно от човека. С кое сетиво става това възприемане?
2. Посочете в природата източници на информация, която не се възприема директно от човека. Как сме достигнали до знанието за съществуване на всеки такъв източник?
3. Съществуват ли природни източници на информация, за които не подозираме?
4. Запознайте се по-подробно с работата на родителите си или на други близки. Посочете източници на информация на тяхното работно място.
5. Вгледайте се във вашето ежедневие. Посочете източници на информация, която представляват интерес за вас. Как добивате необходимата ви информация от тези източници?
6. Можете ли да откриете някаква информация във всяко от следните изображения:
 - а) 
 - б) 
 - в) 
 - г) 
7. Можете ли да се открие някаква информация в шума на работещ на високи обороти автомобилен двигател?
8. Можете ли (без да използвате специална техника), да откриете някаква информация в течност, налята в чаша, която няма никакъв цвят и миризма?
9. Има ли информация:
 - а) в романа „Под игото“;
 - б) в стихотворението „Две хубави очи“;
 - в) в рапсодия „Вардар“;
 - г) в изображението на Мадарския конник;
 - д) в предпочитания от Вас видеоклип на любима група?
10. Посочете информация, която си струва да бъде съхранена. Защо смятате, че такава информация е полезна?
11. Посочете информация, която не си струва да бъде съхранена. Защо смятате, че такава информация е безполезна?
12. Посочете информация, която трудно може да бъде използвана, ако е съхранена по класически начин – на хартия.

2

Компютри и програми

В този урок ще припомним накратко изученото в часовете по Информационни технологии за *компютрите и програмите*.

Първи опити за автоматизиране на пресмятанията

Още от дълбока древност човечеството е познавало и използвало различни технически средства за пресмятане. Първите такива средства са *абациите*, създадени преди повече от 1500 г. Техните съвременни аналози – *металата* – се използват и до днес.

Истинска революция в автоматизирането на пресмятаня е създадената от френския математик Блез Паскал (*Фиг. 1а*) *аритметична машина* или *аритмометър*, с която много бързо се събират големи



а



б

Фиг. 1. Блез Паскал (а) и неговият аритмометър (б)

числа (Фиг. 1б). Великият немски математик, Годфрид Вилхелм Лайбниц, прави съществени подобрения в аритмометъра на Паскал, за да може да извършва бързо четирите аритметични операции – събиране, изваждане, умножение и деление. Усъвършенствани варианти на аритмометрите са съвременните *електронни калкулатори*, в които всички механични и електро-механични части са заменени с електронни схеми.

Машини с програмно управление

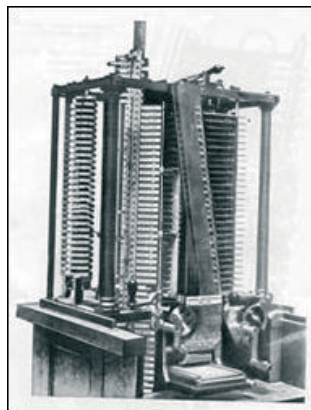
Важна стъпка към създаването на съвременния компютър прави френският инженер Жозеф-Мари Жакар, създал *тъкачен стан с програмно управление*. **Принципът за програмно управление** е от съществено значение и затова ще го формулираме по-точно:

Машина с програмно управление наричаме устройство, което изпълнява множество операции и е снабдено с механизъм за задаване на различни последователности от такива операции – *програми*. Машината с програмно управление следва указанията на програмата и автоматично изпълнява предписаните в нея операции.

Идеята да се създаде математическа машина, която да **изпълнява автоматично операции, зададени с програма**, принадлежи на английския учен-енциклопедист Чарлз Бабидж (Фиг. 2а). През 1833 г. той разработва *аналитичната машина* (Фиг. 2б), съчетание на аритмометър с програмно устройство. Близката сътрудничка на Ч. Бабидж, графиня Огъста-Ада Лъвлейс (Фиг. 2в), дъщеря на поета Байрон, само на основата на описанието на машината започнала да съставя програми за бъдещи изчисления, поставяйки началото на професията *програμισ*.



а. Чарлз Бабидж



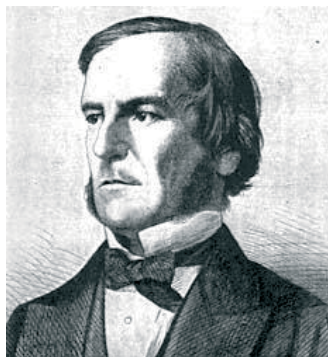
б. аналитична машина



в. лейди Ада Лъвлейс

Фиг. 2.

В средата на 30-те и началото на 40-те години на миналия век усилията за създаване на **универсална програмируема машина за автоматизирано изпълнение на алгоритми** се увенчават с успех. Първата стъпка е разработването от Клод Шенън на **електро-механични устройства за извършване на аритметични операции**, основани на създадена от Джордж Бул (Фиг. 3а) в края на XVIII век *двоична логика*.



а. Джордж Бул



б. Джон Атанасов



в. Машината ABC

Фиг. 3.

През януари 1939 г. американецът от български произход Джон Атанасов (Фиг. 3б), съвместно с Клифърд Бери, създава **първия реално действащ модел** на компютър – ABC (Фиг. 3в). Приносът на Атанасов е огромен, защото вместо електро-механичните елементи на Шенън, в ABC се използва **електроника**.

Принципи на фон Нойман

Значителна роля за оформяне на облика на съвременния компютър изиграва Джон фон Нойман. Работейки заедно с колегите си Екерт и Моучли върху машината EDVAC, Джон фон Нойман (Фиг. 4) обосновава принципите на съвременния компютър, известни като *принципи на фон Нойман*. Компютрите, построени на тези принципи, наричаме *фон Нойманови компютри*.

Накратко ще формулираме принципите, които фон Нойман обобщава и интегрира в цялостна концепция:

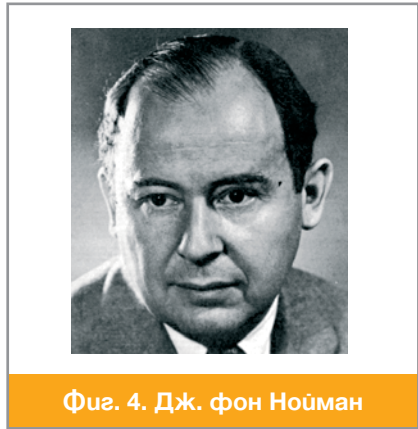
Първи принцип. Компютърът е електронно устройство. Всички операции в него се извършват от електронни схеми.

Втори принцип. Компютърът е двоично устройство. Данните се съхраняват в компютъра в двоичен вид. Електронните схеми също са двоични.

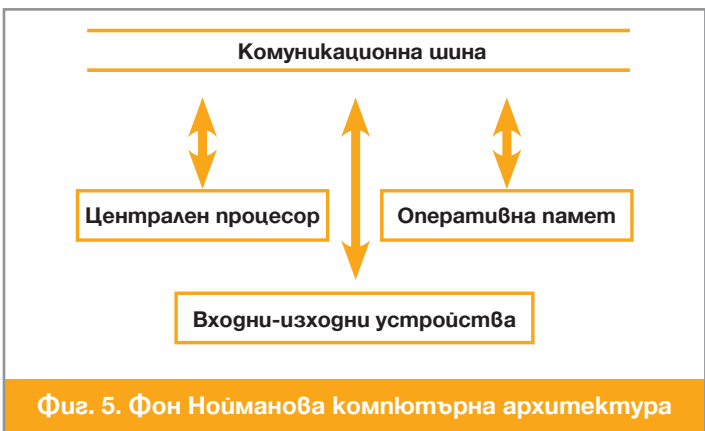
Трети принцип. Компютърът е управляемо от програма устройство. Програмата и входните данни се въвеждат в *компютърната памет* и се съхраняват там по време на работа. Програмите се състоят от *инструкции*, които се извличат в определен ред в *компютърния процесор*, изпълняват се над данните и резултатите се връщат обратно в паметта.

Машината EDVAC (Джон Екерт и Джон Моучли) е първата построена изцяло в съответствие с принципите на фон Нойман и от нейното название идва думата *компютър*.

Съвременните компютри са построени по една и съща принципна схема, наречена *архитектура на фон Нойман*. Тази схема (показана на Фиг. 5), както и предназначението на Централния процесор, Оперативната памет и Входно-изходните устройства познаваме от уроците по ИТ. Ще се спрем по-подробно на начина на тяхното функциониране в този и следващи уроци.



Фиг. 4. Дж. фон Нойман



Фиг. 5. Фон Нойманова компютърна архитектура

Съгласно Третия принцип на фон Нойман, компютърът е машина с програмно управление. Устройството, което осъществява програмното управление на компютъра, е Централният процесор (ЦП).

ЦП изпълнява *инструкции*, всяка от които предписва някаква *операция* с данни от оперативната памет (ОП). *Кодът* на инструкцията определя операцията, а *аргументите* – данните, с които да се извърши предписаната операция. ЦП извлича инструкцията и зададените в нея аргументи от ОП, стартира схемата за извършване на операцията, връща резултата в ОП и определя следващата инструкция. Системата от инструкции на компютъра и правилата за изпълнението им наричаме *машинен език*. Поредица от инструкции наричаме *програма*.

```

1000 СЪБЕРИ X Y
1001 СЪБЕРИ X 5
1002 УВЕЛИЧИ Y
1003 ПРЕМЕСТИ X Y
1004 КРАЙ
    
```

	...
X	1000
	...
Y	8
	...

Фиг. 6. Програма

```

ПРИ=0 X A
ПРИ>=0 X A
СКОЧИ A
    
```

Фиг. 7. Инструкции за преход

```

1000 СЪБЕРИ X Y
1001 ПРИ>=0 X 1003
1002 УМНОЖИ X -1
1003 ПОКАЖИ X
1004 КРАЙ
    
```

Фиг. 8.

На Фиг. 6 е показана програма от четири инструкции и част от ОП. За удобство, двоичните кодове са представени с по-разбираеми означения. След изпълнението на всяка от тези инструкции ЦП преминава към изпълнение на следващата инструкция.

Първата инструкция предписва да се съберат X и Y, а резултатът да се съхрани в X. В резултат, съдържанието на X ще стане 1008 и ЦП ще премине към втората инструкция. Тя прибавя 5 към съдържанието на X и то става 1013. Третата инструкция увеличава съдържанието на Y с единица, а четвъртата – заменя съдържанието на X с това на Y. Какви ще бъдат стойностите на X и Y, след изпълнението на всяка от тези инструкции? Инструкцията **КРАЙ** е указание за процесора да преустанови работата на програмата.

Инструкциите се разполагат в ОП, както и данните. Така всяка инструкция получава *адрес* в паметта. Числото пред всяка инструкция на Фиг. 6 е нейният адрес. Всеки машинен език има *инструкции за разклоняване* или *преход*. На Фиг. 7 са показани примери на такива инструкции. Инструкцията с код **ПРИ=0** предписва да се провери съдържанието на първия аргумент и ако е 0, ЦП трябва да премине към изпълнение на инструкцията с адрес във втория аргумент. В противен случай, т.е. X не е нула, ЦП трябва да продължи със следващата инструкция на програмата в ОП. **Когато ЦП изпълни инструкция, която не е за преход, винаги продължава със следващата в ОП инструкция.**

Възможни са и по-сложни проверки, както в **ПРИ>=0**, която извършва прехода, когато стойността на първия аргумент е по-голяма или равна на 0. Инструкцията **СКОЧИ** предписва преходът да се извърши, без да се проверява условие – *безусловен преход*. На Фиг. 8 е показана програма, която намира и показва на екрана абсолютната стойност на сумата на числата, намиращи се в паметта на адреси X и Y. Проследете работата на програмата, ако X = 3 и Y = -7.

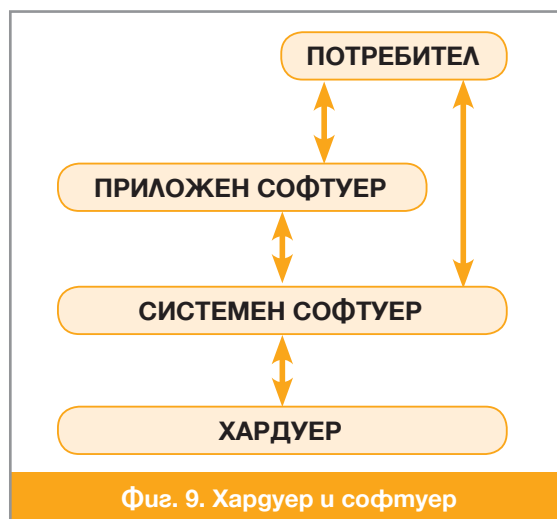
Машинният език е труден за усвояване и неудобен за всекидневна употреба. Затова в масовото производство на програми се използват *езици за програмиране*, които са по-близки до човешкия. Един от най-популярните езици за програмиране е езикът C (четем „си“). От него са произлезли различни по-модерни езици, включително и езикът C# (четем „си шарп“), с който ще се запознаем в уроците по информатика.

И така, компютърът е сложно устройство (*хардуер*), изпълняващо програми, които решават интересни за потребителя задачи за съхраняване, обработка и изобразяване на данни. Програмите изпълнявани от

един компютър, наричаме *софтуер*. Те могат да се разделят на две групи, в зависимост от кръга потребители, за които представляват интерес. Едни програми са полезни за всеки потребител – *системен софтуер*, а други само за отделни потребители – *приложен софтуер* (Фиг. 9).

В системния софтуер също могат да се обособят две групи програми. В първата група са няколко взаимно свързани програми, наречани *операционна система* (ОС). От уроците по ИТ познаваме предназначението на ОС, оновните ѝ функции и знаем как да си служим с ОС в ежедневната си работа с компютъра.

В другата група програмите са независими една от друга и се наричат *инструментални програми* (utilities). Може да се каже, че инструменталните програми са приложни програми, които решават сериозни задачи, подпомагат работата предимно на квалифицирани специалисти в областта на информатиката и затова се считат за част от системния софтуер. В следващ урок ще започнем разглеждането на един основен за информатиката клас инструментални средства – *интегрираните среди за разработване на софтуер*.



Въпроси и задачи

1. Защо идеята на Жакард за програмно управление е повратен момент в историята на създаването на компютрите?
2. Кой за пръв път споменава за *двоична* математическа теория, оперираща само с две стойности?
3. На кого принадлежи идеята за построяване на изчислителни устройства на електро-механичен принцип?
4. Кой е авторът на първия проект за електронни устройства, извършващи математически изчисления?
5. Коя е първата изчислителна машина, построена изцяло в съответствие с принципите на фон Нойман?
- 6*. Кой от принципите на фон Нойман биха могли да се променят в резултат на развитието на науката и технологиите?
7. Защо наричат ЦП „мозък“ на компютъра?
- 8*. Напишете програма на машинен език, пресмятаща изразите:

а. $A + B + C$;	б. $A + B.C$;	в. $A.B + C.D$,
------------------	----------------	------------------

 ако стойностите на A , B , C и D се намират в едноименни полета на паметта, а стойността на израза се запише в поле от паметта, означено с X .
- 9*. Напишете програма на машинен език, пресмятаща верността на условията:

а. $A + B + C < 0$;	б. $A + B > C$;	в. $A.B \neq C.D$,
----------------------	------------------	---------------------

 ако стойностите на A , B , C и D се намират в едноименни полета на паметта, а верността на условието – означена с 0 за неистина и 1 за истина – се записва в поле от паметта, означено с X .

Позиционни бройни системи

За да можем да извършваме пресмятания с числа, те трябва да бъдат представени по подходящ начин. Една съвкупност от правила за представяне на *естествените числа* наричаме *бройна система*,

тъй като с естествените числа означаваме броя на някакви обекти – 0, 1, 2, ..., 397, и т.н. Всяка бройна система си служи с няколко *цифри*. Числовите системи делим на *непозиционни* и *позиционни*.

В *непозиционните* бройни системи стойността на всяка цифра е постоянна и не зависи от нейното място в числото. Пример за такава бройна система е *римската*. В нея цифрата I има стойност 1, цифрата V – стойност 5, цифрата X – стойност 10, и т.н. Стойността на числото е сумата от всички цифри в представянето. Числото 12 се представя в римската бройна система като XII, защото XII = X + I + I = 10 + 1 + 1 = 12. Когато, обаче, цифра с по-малка стойност се постави преди цифра с по-голяма стойност, тогава по-малката се изважда от по-голямата, а не се прибавя. Затова, например, IV = -1 + 5 = 4.

Бройните системи са *позиционни*, когато стойността на цифрата в числото, зависи от това, на коя позиция се намира тя в представянето. За пример да разгледаме десетичната бройна система с цифри 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0. Числото 10 е нейна *основа*. Позициите на числото, наричани още *разряди*, са номерирани отляво наляво с 0, 1, 2, и т.н. Например, в числото 542 цифрата 2 в нулевия разряд има стойност $2 = 2 \cdot 10^0$, докато в числото 1245 тя е във втория разряд и има стойност $200 = 2 \cdot 10^2$. Т.е., ако цифрата c е в i -тия разряд стойността ѝ е $c \cdot 10^i$. Стойността на числото, представено в позиционна бройна система, е сума от стойностите на всичките му цифри. Така $1245 = 1 \cdot 10^3 + 2 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$.

Десетичната бройна система не е подходяща за използване в компютрите, тъй като технически е много трудно да се представят десетичните цифри. Естествено е в електронните устройства да се използва *двоична позиционна бройна система*. В нея, вместо 10 се използват само 2 цифри – 0 и 1, които лесно се представят с отсъствие и наличие на електрически ток, или пък с ненамагнитеност и намагнитеност. Както знаем от уроците по Информационни технологии, разрядите на числата, представени в двоична система, наричаме *битове*. Стойността на всяка цифра в двоична система се умножава по съответните степени на основата 2. Така двоичното число $b_n b_{n-1} \dots b_{0(2)}$ е равно на $N = b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \dots + b_0 \cdot 2^0$.

Всяко число $p > 1$ може да стане основа на бройна система. Други, често използвани в информатиката позиционни бройни системи са осмичната и шестнадесетичната.

Преминаване от двоична система в десетична и обратно

Преминаването от двоична в десетична система става по формулата, дадена по-горе. За целта е добре да имаме, предварително пресметнати, стойностите на степените на основата 2. Първите няколко от тях са дадени в следната таблица:

Степенен показател	7	6	5	4	3	2	1	0
Степен на двойката	128	64	32	16	8	4	2	1

Да преобразуваме представеното в двоична позиционна бройна система число $10101011_{(2)}$ в десетична система. За по-лесно да поставим цифрите на числото в таблица, срещу съответните им степени на двойката:

Двоично число	1	0	1	0	1	0	1	1
Степен на двойката	128	64	32	16	8	4	2	1

Умножаваме всеки бит на двоичното число със съответната степен на двойката:

$$\begin{aligned} 10101011_{(2)} &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = \\ &= 1 \cdot 128 + 0 + 1 \cdot 32 + 0 + 1 \cdot 8 + 0 + 1 \cdot 2 + 1 \cdot 1 = \\ &= 128 + 32 + 8 + 2 + 1 = 171_{(10)}. \end{aligned}$$

А как да преминем от десетична в двоична бройна система? Забележете, че ако разделим числото $N = b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$ на 2, ще получим в резултат цялото частно $b_n \cdot 2^{n-1} + b_{n-1} \cdot 2^{n-2} + \dots + b_1 \cdot 2^0$ и остатък b_0 – най-младшият разряд на представянето на N в двоична система. Получаваме следната

Процедура. Дадено: Естествено число N в десетична бройна система. Действия: Делим числото N на 2 и записваме полученото частно и остатък. Делим остатъка на две и отново записваме полученото частно и остатък. Продължаваме по същия начин, докато получим частно 0. Резултат: Получените остатъци, записани в ред, обратен на реда на получаването им, са двоично представяне на N .

На *Фиг. 1а* е показан пример, в който процедурата е приложена върху числото $171_{(10)}$. Резултатът, разбира се, е $10101011_{(2)}$.

$$\begin{aligned}
171:2 &= 85 \text{ и остатък } 1 \\
85:2 &= 42 \text{ и остатък } 1 \\
42:2 &= 21 \text{ и остатък } 0 \\
21:2 &= 10 \text{ и остатък } 1 \\
10:2 &= 5 \text{ и остатък } 0 \\
5:2 &= 2 \text{ и остатък } 1 \\
2:2 &= 1 \text{ и остатък } 0 \\
1:2 &= 0 \text{ и остатък } 1 \\
171_{(2)} &= 10101011_{(2)}
\end{aligned}$$

$$\begin{aligned}
0 \mid 6015625 \times 2 &= \\
1 \mid 203125 \times 2 &= \\
0 \mid 40625 \times 2 &= \\
0 \mid 8125 \times 2 &= \\
1 \mid 625 \times 2 &= \\
1 \mid 25 \times 2 &= \\
0 \mid 5 \times 2 &= \\
1 \mid 0
\end{aligned}$$

а

б

Фиг. 1. Превръщане от десетична в двоична система

Дробни числа

Когато дробно число е представено в десетична позиционна бройна система, тогава всяка от цифрите му след десетичната запетая се умножава по съответната отрицателна степен на 10. Например $0,1245_{(10)} = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 4 \cdot 10^{-3} + 5 \cdot 10^{-4}$. Аналогично за двоичната система – само че в този случай всяка цифра се умножава по съответната отрицателна степен на 2. Да превърнем дробното число $0,1001101_{(2)}$ от двоична в десетична система:

$$\begin{aligned}
0,1001101_{(2)} &= 1 \cdot 2^{-1} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-7} = 1/2 + 1/16 + 1/32 + 1/128 = \\
&= 0,5 + 0,0625 + 0,03125 + 0,0078125 = 0,6015625_{(10)}.
\end{aligned}$$

Преминаването обратно, от десетична в двоична система, при дробните числа става с умножаване по 2. Забележете, че когато умножим по 2 число, по-малко от единица, първата му двоична цифра след двоичната запетая става цялата част на получения резултат. Така получаваме следната процедура. Умножаваме дробната част на числото по 2. Отделяме получената цяла част – 0 или 1 – като поредна цифра на двоичното представяне и отново умножаваме само дробната част на резултата по 2. Когато получим в резултат дробна част 0, процедурата се прекратява. За някои числа, обаче, това никога не се случва, защото в двоична система те са безкрайни периодични дробни. В такъв случай прекратяваме процедурата, когато намерим достатъчно много цифри след двоичната запетая. На Фиг. 1б е показано преобразуването на числото $0,6015625_{(10)} = 0,1001101_{(2)}$.

Аритметични действия с двоични числа

Процедурите за извършването на аритметични действия с естествени числа, записани в двоична бройна система по нищо не се различават от познатите ни процедури за извършване на аритметични действия в десетична система. Достатъчно е само да знаем съответните таблици за събиране, умножение и изваждане. В двоичната бройна система тези таблици са много по-прости (виж Фиг. 2).

Събиране	Изваждане	Умножение
$0 + 0 = 0$	$0 - 0 = 0$	$0 \times 0 = 0$
$1 + 0 = 1$	$1 - 0 = 1$	$1 \times 0 = 0$
$0 + 1 = 1$	$1 - 1 = 0$	$0 \times 1 = 0$
$1 + 1 = 0$ и пренос 1	$10 - 1 = 1$	$1 \times 1 = 1$

Фиг. 2. Таблицы за събиране, изваждане и умножение в двоична система

Да напомним, че ако при събиране на цифри получим число, по-голямо от основата, тогава в резултата записваме последната цифра на числото, а останалите цифри образуват число, което наричаме *пренос*. Когато при изваждане поредната цифра на умалителя е по-голяма от съответната цифра на умаляемото, трябва да вземем една единица от най-близкия отляво ненулев разряд. Така че, освен правилото $10 - 1 = 1$, трябва да се имат предвид и правилата $100 - 1 = 11$, $1000 - 1 = 111$ и т.н.

Ще припомним тук само как събираме числа в позиционна система:

Процедура. Дадено: Двоични числа $A = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_0 \cdot 2^0$ и $B = b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + \dots + b_0 \cdot 2^0$ (ако едно от числата има по малко цифри, попълваме го отляво с нули). Действия: В началото преносът q е нула. Обработваме разрядите от дясно наляво, започвайки от нулевия. Когато сме в i -тия разряд, пресмятаме $C = q + a_i + b_i$. Последната цифра на C , да я означим с c_i , записваме в резултата, а преноса запомняме в q . Когато обработим и последния разряд, ако преносът е ненулев – записваме го най-вляво в резултата като c_{n+1} . **Резултат:** $c_{n+1} \cdot 2^{n+1} + c_n \cdot 2^n + c_{n-1} \cdot 2^{n-1} + \dots + c_0 \cdot 2^0$ е сумата на A и B .

На *Фиг. 3* са показани примери за събиране, изваждане и умножение на две числа в двоична бройна система. За упражнение извършете самостоятелно тези действия.

$\begin{array}{r} 11111 \\ + 1110010 \\ \hline 101001101 \end{array}$	$\begin{array}{r} \dots \\ 11011011 \\ - 1110010 \\ \hline 1101001 \end{array}$	$\begin{array}{r} 11011011 \\ \times 1010 \\ \hline 110110110 \\ 110110110 \\ \hline 100010001110 \end{array}$
<p>Фиг. 3. Събиране, изваждане и умножение в двоична система</p>		

Сравняването на числа в позиционна бройна система – независимо от основата, извършваме със следната:

Процедура. Дадено: Двоични числа $A = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_0 \cdot 2^0$ и $B = b_m \cdot 2^m + b_{m-1} \cdot 2^{m-1} + \dots + b_0 \cdot 2^0$, $a_n \neq 0$ и $b_m \neq 0$. Действия: Ако $n > m$, тогава $A > B$. Ако $n < m$, тогава $A < B$. Ако $n = m$, тогава сравняваме цифрите на двете числа в едни и същи разряди, започвайки от най-левия. Ако не намерим различаващи се цифри в нито една от позициите, значи $A = B$. Нека цифрите в i -тия разряд са различни. Ако $a_i > b_i$, тогава $A > B$. В противен случай $A < B$.

Осмична и шестнадесетична бройна система

Освен двоичната система в информатиката се използват *осмичната* и *шестнадесетичната* система. Осмичната система използва десетичните цифри от 0 до 7. Но за бройна система с основа по-голяма от 10, трябва да се добавят знаци, за които няма десетични цифри. В шестнадесетична система за цифри, съответни на 10, 11, 12, 13, 14 и 15, се използват латинските букви А, В, С, D, Е и F. Например, стойността на числото $1AE_{(16)}$ в десетична бройна система, е:

$$1AE_{(16)} = 1 \cdot 16^2 + 10 \cdot 16^1 + 14 \cdot 16^0 = 256 + 160 + 14 = 430_{(10)}$$

Осмичната и шестнадесетичната система са характерни с това, че много лесно се преминава от тях в двоична система и обратно. Преминаването от двоична в осмична система става като разбием цифрите на двоичното представяне на тройки отдясно наляво – ако цифрите не са достатъчни за пълна тройка, тогава добавяме необходимия брой нули отляво. След това всяка тройка от двоични цифри замества със съответното осмично число, както са дадени в следната таблица:

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

За преминаване от осмична в двоична система пък е достатъчно да заменим всяка осмична цифра със съответната тройка двоични цифри. Например $1111110010_{(2)} = (001)(111)(110)(010) = 1762_{(2)}$, $3407_{(8)} = (011)(100)(000)(111) = 11100000111_{(2)}$.

Преминаването от двоична в шестнадесетична система и обратно е аналогично, като всяка шестнадесетична цифра се заменя със съответната ѝ четворка двоични цифри от таблицата

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

Въпроси и задачи

1. Каква е разликата между позиционна и непозиционна бройна система?
2. Кое десетично число е $1101_{(2)}$? А $1101_{(8)}$ и $1101_{(16)}$?
3. Намерете стойността на:
 - а) $31_{(10)}$ в двоична бройна система;
 - б) $12_{(8)}$ в двоична бройна система;
 - в) $32_{(10)}$ в осмична бройна система;
 - г) $С8_{(16)}$ в двоична бройна система.
4. Нека $A=100110_{(2)}$ и $B=11011_{(2)}$. Намерете:
 - а) сумата на A и B ;
 - б) разликата на A и B ;
 - в) произведението на A и B .
5. Дадено е числото $10111_{(2)}$. Кое е следващото по големина естествено число?
6. Превърнете числото $0,2005_{(10)}$ в двоична система, а числото $0,1101_{(2)}$ – в десетична.
7. Сравнете по-големина числата A и B :
 - а) $A=100110_{(2)}$ и $B=11011_{(2)}$;
 - б) $A=1001101101110110_{(2)}$ и $B=10011011011100110_{(2)}$.

4

Съхраняване на данни в компютъра

За съхраняването на данните в оперативната памет на компютъра се използват само естествени числа, представени в двоична позиционна бройна система. В този урок ще покажем как само с естествени числа може да се представи цялото разнообразие от данни, съхранявани в компютърната памет – текстове, цели и дробни числа, изображения и т.н. Ще си припомним изучаваното в уроците за Електронни таблици понятие *тип на данните*.

Мерни единици за обема на данните

Да си припомним наученото за мерните единици от уроците по ИТ. Най-малката мерна единица за обема на данните съхранени в компютърната памет, е *битът* (bit, от **binary digit** – двоична цифра). В един бит можем да запомним една двоична цифра – 0 или 1. Осем последователни бита образуват *байт*. Тъй като всеки бит на байта може да заема стойност 0 или 1, в един байт можем да запишем $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^8 = 256$ различни стойности Байтът (B) е най-малкото поле от ОП, което има свой адрес. За обозначаване обема на големи съвкупности от данни се използват по-големи мерни единици.

Един кибибайт (KiB) съдържа $1024 = 2^{10}$ байта, един мебибайт (MiB) – 1024 KiB или 2^{20} байта, един гигабайт (GiB) – 1024 KiB или 2^{30} байта и т.н.

Тип на данните

Ако можем да разгледаме едно поле от ОП, щяхме да видим в него записана последователност от нули и единици. Каква данна е записана там зависи само от начина, по който се тълкува съдържанието на полето. Начинът по който компютърът тълкува записаното в едно поле на паметта, наричаме (*базов или реален*) *тип на данните*. Всеки компютър има заложени в себе си няколко базови типа данни. Характерно за всеки базов тип е *размерът*, (в байтове), на полето от ОП, който всяка данна от този тип заема в паметта, както и *операциите*, които могат да се изпълняват с данните от този тип. Размерът на типа и начинът, по който се представят данните в полето, еднозначно определят множество от *възможни стойности* на данните.

Естествено е, че не можем да очакваме компютърът да предоставя всевъзможни типове данни, които ще ни се наложи да използваме при създаването на компютърни приложения. Затова всеки език за програмиране предвижда механизъм, по който от базовите типове да се създават нови – *потребителски* или *абстрактни типове* данни. В следващите раздели на урока ще покажем основните базови типове и как с **комбинация от базови типове** може да се дефинира необходимият ни потребителски тип.

Естествени и цели числа

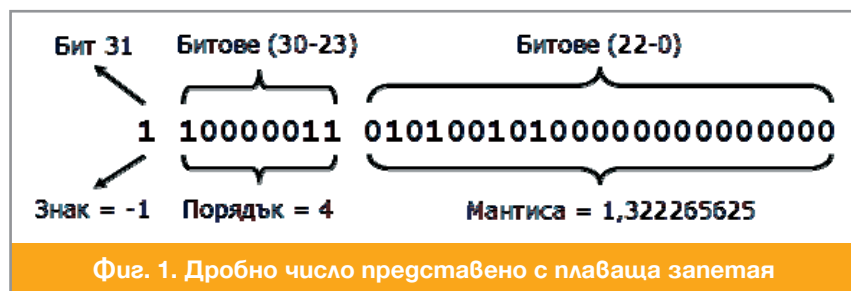
Естествените числа се представят в компютърната памет както в разгледаната в предишен урок двоична бройна система, с тази разлика че полето, предназначено за числото, се попълва отляво с нули. Да разгледаме, като пример, как се представят естествени числа само в един байт. В байт може да се запише в двоична бройна система всяко естествено число с не повече от 8 двоични цифри – т.е. числата от $00000000_{(2)} = 0$ до $11111111_{(2)} = 255$. Такива типове, при които стойностите са естествени числа, наричаме в програмирането „беззнакови“ (unsigned).

За да можем да представяме цели числа – както положителни, така и отрицателни, е необходимо да различаваме едните от другите. За целта трябва да отделим един от битовете на полето, за да показва положително или отрицателно е числото. Това обикновено е старшият бит. Когато числото е положително, в старшия бит се записва нула, а когато е отрицателно – единица. В оставащите 7 бита могат да се запишат естествените числа от 0 до $1111111_{(2)} = 127$. За представяне на отрицателните числа има различни методи. Един от тях е така нареченият *прям код* – в седемте бита на отрицателно число се записва абсолютната му стойност, но тогава има две различни представяния на нулата, което не е удобно. Реално се използват кодове, които водят до представяне на отрицателните числа от -1 до -128 .

И така, в един байт може да се запишат числа без знак в интервала от 0 до 255 или числа със знак, в интервала от -128 до 127. В целочислен тип със знак с размер 2 байта могат да се запишат числа без знак в интервала от 0 до 65535 или числа със знак, в интервала от -32768 до 32767 и т.н.

Дробни числа

За представяне на дробно число R в компютърната памет се използва неговият *експоненциален вид*: $R = M \cdot q^p$, където M е *мантиса* на R , p е *порядък* (или *експонента*), а q е основа на бройната система, в която е представено числото. За представяне в компютърната памет на дробни числа се използват само две основи – 2 или 10. Мантисата е положителна или отрицателна правилна дроб, за която е изпълнено $R = M \cdot q^p$. Порядъкът е положително или отрицателно цяло число. Например, едно експоненциално десетично представяне на числото $-32,87$ е $-0,3287 \cdot 10^2$ и в този случай $M = -0.3287$ и $p = 2$, а едно такова представяне на числото $0,000152$ е $1,52 \cdot 10^{-4}$, $M = 1,52$ и $p = -4$. Тъй като в прост текст не е възможно степените да се изписват умалени и малко по-високо, в информатиката е прието в експоненциалния запис на числото да се замества основата с буквата E , а вместо десетична запетая се изписва десетична точка. Затова горните две експоненциални представяния се записват като $-0.3287E2$ и $1.52E-4$.



За представяне на дробните числа в компютъра се използват 3 полета: s – за знака на числото, e – за порядъка и w – за абсолютната стойност на мантисата. Полето s винаги е от 1 бит и в него се записва 0, когато числото е положително, или 1 – когато е отрицателно. От всички

възможни абсолютни стойности на мантисата се избира тази M , за която $1 \leq M < 2$, а тя определя и съответния порядък p . В полето w се записва дробната част на M , тъй като цялата част е винаги 1 и се подразбира. Записът на порядъка е естественото число $p' = p + \text{max}$, където max е подходяща константата. Такова представяне на дробното число се нарича представяне с *плаваща запетая* (floating point).

Да разгледаме един пример за представяне на число с плаваща запетая в паметта в 32 бита (*single precision*) съгласно популярния стандарт IEEE-754. При този формат се използват 23 бита за мантисата и 8 бита за порядъка, а $emax = 127$ (Фиг. 1).

Знакът на числото е отрицателен. Абсолютната стойност на мантисата е:

$$M = 1,010100101_{(2)} = 1 + 2^{-2} + 2^{-4} + 2^{-7} + 2^{-9} = 1 + 1/4 + 1/16 + 1/128 + 1/512 = 1 + 0,25 + 0,0625 + 0,0078125 + 0,001953125 = 1,322265625_{(10)}$$

Кодът на порядъка е $p' = 10000011(2) = 2^7 + 2^1 + 2^0 = 128 + 2 + 1 = 131$. Значи за порядъка получавате $p = p' - emax = 131 - 127 = 4_{(10)}$. И така, представеното на Фиг. 1 дробно число с плаваща запетая е $-1,322265625 \cdot 2^4 = -21,15625$.

Знаци

Освен числа в компютъра най-често съхраняваме данни под формата на текст. За представяне на *знаците*, от които съставяме текстовете, се използват естествени числа. Много години обичайният начин за представяне на знак беше с число от 0 до 255, което се помещава в един байт цял тип. На кой знак кое число съответства се определя от *кодова таблица*. Най-често използвана беше кодовата таблица ASCII, част от която е показана на Фиг. 2.

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
...																
128	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
144	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
160	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
176	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
...																

Фиг. 2. Когова таблица ASCII

В тази таблица задължително участват буквите на латинската азбука и е определено място, където държавите, използващи други азбуки, да разположат своите букви. В българския вариант на таблицата ASCII, на големите букви на кирилицата са съпоставени кодовете от 128 до 159, а на малките букви – от 160 до 191 (включително буквите ы и э, които се използват само в руския език).

Поради голямото разнообразие от използваните в света азбуки, представянето ASCII не е добро решение за редица случаи. Например, при публикуване в Интернет на текстове, съдържащи знаци от няколко различни ASCII таблици – латински букви, букви на кирилица и арабски букви. Затова в бъдеще все по-често ще се използват таблици, в които на знаците се съпоставят по-големи естествени числа – например, двубайтови, трибайтови и дори четирибайтови. Най-популярната днес система за представяне на знаци в компютърната памет е Unicode. Тя съдържа всички широко използвани по света знаци на всички езици. Ще имаме възможност да се запознаем с нея в следващ урок.

Представяне на изображения

От уроците по ИТ знаем, че екранът на компютъра е *растер* – правоъгълна таблица от *пиксели*, всеки от които може да бъде оцветен в различни цветове. Така се получават компютърните изображения. Идентифицирането на всеки пиксел на екрана става с наредена двойка от естествени числа (x, y) – номера x на реда и номера y на стълба на пиксела в растера.

Цветът, в който трябва да бъде оцветен един пиксел, също се задава с числа. Системата RGB, която познаваме също от уроците по ИТ, представя всеки цвят с тройка от еднобайтови цели без знак (r, g, b) , където r е интензитетът на червения цвят в комбинацията от червено, зелено и синьо; g – интензитетът на зеления; b – интензитетът на синия.

По подобен начин, всички останали обекти, от които имаме нужда, се представят в компютърната памет с помощта на числа.

Въпроси и задачи

1. Кое е най-голямото число без знак в десетична бройна система, което може да се запише в:
а) 4 бита? б) 16 бита? в) 32 бита? г) 8 байта?
2. Кое десетично число е записано в байта: 00001101 ?
3. Представете с мантиса и порядъка числото 145,12 в десетична бройна система.
4. На кое реално число експоненциалният запис е $0.2761E-2$?

5

Система за програмиране

Система за програмиране е съвкупност от инструментални програми, предназначени за създаване на нови програми. Системата за програмиране е основният инструмент на програмистите. В една система за програмиране могат да бъдат включени следните компоненти: език за програмиране, транслятор от него, текстов редактор, дебъгер и т.н. В този урок ще разгледаме предназначението на основните компоненти на една система за програмиране.

Машинни езици и асемблери

Основен елемент на системата за програмиране е *езикът за програмиране*. Единственият език, който компютърът „разбира“, е неговият *машинен език*. Първите програми са написани на машинните езици на съответните компютри, съдържащи стотици инструкции, в които кодовете на операциите и адресите на аргументите се изписват в цифров вид. Запомнянето на *синтаксиса* на инструкциите (как се изписва всяка инструкция) и *семантиката* им (какво точно предписва всяка инструкция) е доста трудно. За избягване на грешки се налагат чести справки в описанието на езика. Освен това програмата, написана на машинния език на един компютър, не може да бъде пренесена на друг компютър.

За облекчаване работата на програмиста се създават *асемблерните езици*. Разликата между тях и машинния език е, че в програмите, написани на асемблерен език, се допуска използване на нечислови последователности от знаци за означаване кодовете на операциите и адресите на операндите. Служебните думи ADD, SUB, MULT и DIV (съкращения от английските наименования на аритметичните операции add, subtract, multiply and divide), например, могат да се използват за означаване на аритметичните операции. За областта от паметта, съдържаща обема на конкретен обект, може да се избере името *volume*, за поле, съдържащо възраст – *age* и т.н. В предишен урок, представяйки понятията инструкция и програма, дадохме примери на програми на измислен език, който по вида си е асемблерен.

„Преводът“ от асемблерен на машинен език се извършва от специална програма – *асемблер*, която замества имената на инструкциите и имената на аргументите със съответните кодове и адреси. Асемблерен език, асемблер и *текстов редактор* образуват най-простата среда за програмиране.

Алгоритмични езици

Програмирането на асемблерен език е доста по-лесно от програмирането на машинен език и до днес не е загубило своята роля. Асемблерният език, обаче, носи твърде малко облекчения на програмиста.

С разширяване на сферата на приложение на компютрите, в 60-те години на миналия век се появяват езици, първоначалната цел на които е програмистите да обменят алгоритми помежду си, затова отначало са наричани *алгоритмични*. Алгоритмичните езици са от **по-високо ниво** от асемблерните. Една инструкция (оператор) на езика от високо ниво съответства на няколко машинни инструкции. Тези езици се доближават до човешкия, защото операторите им се съставят от разбираеми за човека фрази (обикновено от английския език). Например, *if... then...* (ако ... тогава ...), или *while... do...* (докато ... прави ...). Програмите, написани на алгоритмичен език, се четат много по-лесно, което го прави

най-доброто средство за обмен на алгоритми и обучение в алгоритмизация на задачи за програмиране. Постепенно алгоритмичните езици изместват асемблерните от програмистката практика и започват да се наричат *езици за програмиране*.

До идеята за такъв език пръв е достигнал Конрад Цузе (*Фиг. 1а*), съзателят на едни от първите реално действащи програмируеми, но не електронни, изчислителни устройства. През 1945 . той проектира езика Plancalcul, който остава незавършен, но редица елементи от този проект се срещат по-късно в много езици за програмиране. През 1954-57 г., под ръководството на Джон Бекъс (*Фиг. 1б*) в IBM е разработен езикът FORTRAN (FORmula TRANslation, превод на формули). Предназначен в началото за бързо програмиране на числени пресмятания, по-късно той се използва като универсален език и оказва голямо влияние върху развитието на езиците за програмиране, независимо от някои свои несъвършенства. Начинът за пресмятане на стойността на математически изрази при зададени стойности на променливите във FORTRAN се среща и в най-съвременните езици за програмиране. Езикът FORTRAN претърпя голяма еволюция, като следващите му версии FORTRAN II, FORTRAN IV, FORTRAN 77, FORTRAN 85 отразяваха развитието на езиците за програмиране, а най-новата му версия FORTRAN 9x се използва и днес.

По същото време, колектив под ръководството на Грейс Хопър (*Фиг. 1в*) работи над специализирани езици за обработка на икономическа информация. Най-сполучливият е езикът COBOL (COmmercial and Business Oriented Language, език за бизнес приложения). Счита се, че 60% от софтуера за управление на бизнеса, който работи и в момента, е написан на COBOL.

В края на 50-те години специален научен комитет се занимава с разработването на принципите на универсален език за програмиране, който да не е ориентиран само към един тип задачи. Създаденият от този комитет език Algol (ALGOrithmic Language, алгоритмичен език) не успява да се наложи в практиката, но оказва голямо влияние върху развитие на езиците за програмиране.

До неотдавна много популярни бяха езиците BASIC (създаден от Джон Кемени и Томас Курц) и Pascal (създаден от Никлаус Вирт). Важен за развитието на езиците за програмиране е езикът C (създаден в началото на 70-те години от Брайън Кернигън и Денис Ричи, *Фиг. 1г* и *Фиг. 1д*), тъй като много от най-използваните днес езици – C++, Java, Perl, C# и др. – са породени от C.



Транслатори

FORTTRAN пожънал огромен успех, защото Джон Бекъс и сътрудниците му създали програма – *транслатор*, която превеждала програмите от FORTRAN на машинен език. По качествата си, получените програми не се различавали много от написаните от опитен програмист. При това програмирането на FORTRAN е много по-бързо и по-надеждно от програмирането на асемблерен език. Разработките на Бекъс предизвикват революция в програмирането.

Различаваме два вида транслатори. Първият вид са *компилаторите*. Компилаторът избира за всяка конструкция на езика за програмиране подходящ, предварително подготвен фрагмент на машинен език, и от всички фрагменти съставя (*компилира*) програма. В резултат на работата на компилатора се получава пълен превод на *изходния текст* (source code) на програмата на машинен език – *изпълнима програма* (executable). По този начин не се налага превеждане на програмата всеки път, когато потребителят пожелае да я изпълни върху конкретни данни.

Свършено различен е подходът при *интерпретаторите*. Те също подбират подходяща последователност от машинни команди за всеки от операторите на езика, но тя се изпълнява веднага (казваме, че операторът се *интерпретира*), след което се преминава към интерпретиране на следващия оператор в програмата. Ако по време на изпълнение на програмата съответният оператор се достигне още веднъж, ще бъде преведен отново. Интерпретатори се създават много по-лесно от компилаторите. Интерпретирането на програмата обаче е много по-бавно от изпълнението на компилирана програма. Като пример ще посочим интерпретатори за многобройните версии на езика BASIC.

Редактор и дебъгер

Основен елемент от създаването на една програма е написването на изходния текст на програмата. Това става с обичайното средство за работа с текстове – *текстовите редактори*. Тъй като добре познаваме тези инструментални програми от уроците по ИТ, няма да се спираме подробно на тях тук. Ще отбележим само съществуването на специализирани текстови редактори за създаване на програми, които предлагат редица специални възможности – оцветяване на отделните части на програмата в различни цветове, подреждане на текста, подсказки за евентуални грешки и т.н.

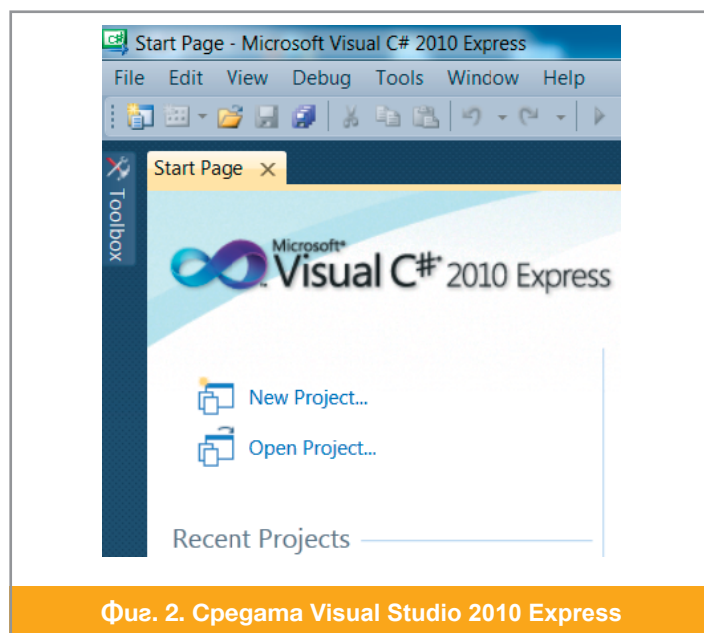
След създаването на програма, която да е работоспособна, не е изключено тя да не дава исканите резултати, т.е. да е логически неправилна. Изключително тежкият етап на проверка на програмата и изчистването ѝ от логическите грешки може да бъде улеснен от специална програма – *дебъгер* (от debug, изчистване на грешки в компютърна програма), позволяваща програмата да се изпълни стъпка по стъпка и така да се намери мястото, където е допусната грешка.

Среда за програмиране

Съвременната тенденция е да се интегрират всички елементи на системата за програмиране (включително и текстов редактор) в едно цяло, наричано *интегрирана среда за програмиране* (Integrated Development Environment или IDE). Така на програмиста се осигурява средство за бързо и лесно създаване на програми. Много често в рамките на една и съща среда са организирани няколко системи за програмиране, всяка от които е свързана с различен език за програмиране.

В този учебник ще се запознаем със средата за програмиране Visual Studio Express Edition 2010 на Microsoft, която поддържа компилатори от няколко езика за програмиране, включително езика C#, на който ще програмираме в процеса на обучение.

Работа с компютър



Фиг. 2. Средата Visual Studio 2010 Express

Ако желаете да инсталирате на своя компютър средата Visual Studio 2010 Express във версията ѝ с компилатор от езика C#, намерете страницата ѝ в Интернет на адрес: www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express. и натиснете бутона INSTALL NOW – ENGLISH, за да започнете. По време на инсталирането ще бъдете попитани дали искате пълните възможности на средата за управление на бази от данни. Това не е необходимо за нашите цели и ако се откажете от тази възможност, ще са необходими около 155 MB дискова памет за инсталацията.

За да сте готови да изпълните следващите задачи за работа с компютър, проверете работоспособността на средата. Отворете менюто start/All programs и щракнете

върху папката Microsoft Visual Studio 2010 Express. От там може да отворите средата Microsoft Visual C# 2010 Express. Тъй като често ще се налага да стартирате средата, удобно е да поставите икона за бързо стартиране на средата (shortcut) върху лентата на задачите. Припомнете си как става това!

След отваряне на средата ще видите в прозореца лентата с познатите ни от други приложни програми падащи менюта File, Edit, Windows и Help, както и менюта със специфични за средата команди (Фиг. 2). Под нея е лентата, съдържаща много инструменти, типични за приложните програми, работещи в MS Windows.

Въпроси и задачи

1. Защо съставянето и проверката на програми на машинен език е бавно и трудоемко?
2. С какво асемблерните езици улесняват работата на програмистите?
3. Какво е предназначението на транслаторите?
4. Сравнете работата на компилатора и интерпретатора.
5. Възможно ли е за даден език за програмиране да има и компилатор, и интерпретатор?
6. По какво езиците за програмиране се различават от естествените езици?
7. Съвременните езици за програмиране са с почти еднакви възможности. Защо не съществува единствен, общоприет и универсален език за програмиране?

6

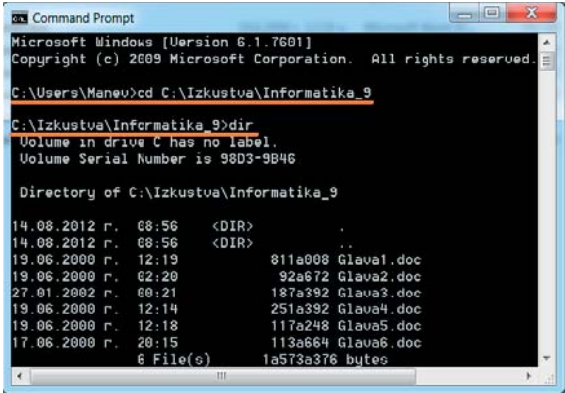
Среда Microsoft Visual C# 2010 Express

В този урок ще се запознаем със средата за програмиране Microsoft Visual C# 2010 и ще напишем първата си програма.

Графичен и буквено-цифров интерфейс

При работа с компютъра потребителят взаимодейства както със системния софтуер (най-вече ОС), така и с приложните програми, с помощта на езици, наричани *интерфейси* (interface). От уроците по ИТ познаваме добре *графичния интерфейс* на операционната система MS Windows и приложните програми, както и неговите елементи – *прозорци* (работни и диалогови), *командни бутони*, *кутии за задаване на параметри* и т.н.

За нуждите на изучаваното в следващи уроци е добре да се познава и другата възможна форма на интерфейс с ОС и приложните програми – *буквено-цифровият интерфейс*. При буквено-цифровия интерфейс, както и при графичния, взаимодействието между потребителя и ОС става с подаване на *команди* и настройване работата на тези команди, посредством *параметри*. При графичния интерфейс подаваме командите с натискане на командни бутони и настройваме параметрите в диалогови прозорци. При буквено-цифровия интерфейс командите са последователности от знаци на клавиатурата, завършващи със знака за нов ред (клавишът Enter). С интервали, командата се разделя на



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Maneu>cd C:\Izkustva\Informatika_9


C:\Izkustva\Informatika_9>dir
Volume in drive C has no label.
Volume Serial Number is 98D3-9B46


Directory of C:\Izkustva\Informatika_9

14.08.2012 г. 08:56 <DIR> .
14.08.2012 г. 08:56 <DIR> ..
19.06.2000 г. 12:19      811a008 Glava1.doc
19.06.2000 г. 02:20      92a672  Glava2.doc
27.01.2002 г. 00:21      187a392  Glava3.doc
19.06.2000 г. 12:14      251a392  Glava4.doc
19.06.2000 г. 12:18      117a248  Glava5.doc
17.06.2000 г. 20:15      113a664  Glava6.doc
               6 File(s)      1a573a376 bytes
```

Фиг. 1. Конзола

отделни думи. Първата дума на командата е нейният идентифициращ *код*, а останалите думи са параметрите ѝ. Затова командите към ОС много приличат на машинните инструкции и образуват език, който се нарича *команден език*.

За да подаваме команди към ОС, трябва да отворим прозорец за буквено-цифров интерфейс, наричан още *конзола* (Фиг. 1). За целта трябва да натиснем бутона **Start** и от менюто **All programs/Accessories** да изберем  **Command Prompt**. На Фиг. 1 конзолата е показана след изпълнението на две команди (подчертани с червено). Първата команда е `C:\Users\Manev>cd C:\Izkustva\Informatika_9`. Всеки команден ред започва с името на текущата папка, в случая `C:\Users\Manev` и знакът `>`, който означава, че ОС очаква въвеждането на команда. Самата команда е `cd C:\Izkustva\Informatika_9`. Кодът на командата `cd` (*change directory*, смени папката) показва, че искаме да сменим текущата папка със зададената като параметър `C:\Izkustva\Informatika_9`. В резултат на това, както се вижда от следващата команда `C:\Izkustva\Informatika_9>dir` текуща папка вече е зададената в командата `cd`. Командата `dir`, без аргументи, предизвиква извеждане на съдържанието на текущата папка на конзолата.

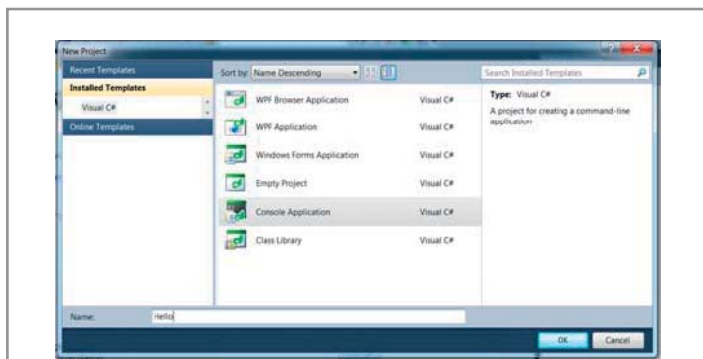
 **Задача.** Отворете конзола на компютъра, на който работите, изберете пака от файловата система, преместете се в нея като в текуща и разпечатайте съдържанието ѝ.

Проект и решение. Конзолно приложение

Важни за работата в средата **Microsoft Visual C# 2010 Express** (за да пестим място, от тук нататък ще я наричаме просто **средата**) са понятията *проект* и *решение*. За създаването на една компютърна програма обикновено са нужни няколко различни файла, създавани от програмиста, както и служебни файлове на средата, с помощта на която се управлява създаването на програмата. Съвкупността от всички тези файлове, се нарича *проект* (*project*).

От файловете на проекта, след подходяща обработка, се получава резултат, който се нарича *решение* (*solution*). Решението може да бъде изпълнима програма, библиотека от подпрограми, която да бъде използвана в текущия или други проекти и т.н. Едно решение може да е съставено от няколко различни проекта. Най-просто за създаване решение е изпълнима програма с буквено цифров интерфейс – *конзолно приложение* (*console application*). Затова ще започнем със създаване на такова решение.

Работа с компютър

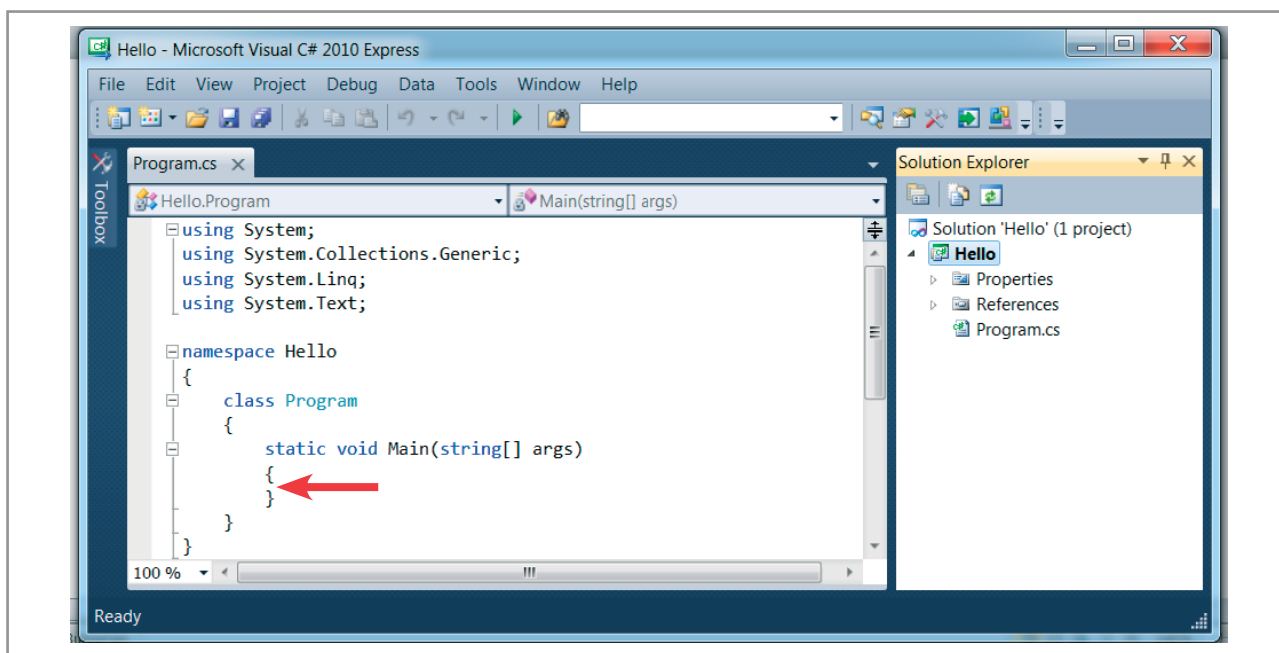


Фиг. 2. Създаване на конзолно приложение

Стартирайте средата. В стартовия прозорец, непосредствено под лентата с инструменти щракнете върху текста **New Project**. След това от диалоговия прозорец изберете **Console Application** (Фиг. 2). Същата команда може да изпълните, като от менюто **File** изберете подменюто **New Project** и от него **Console Application**. Преминаването през няколко менюта, докато стигнем до необходимата ни команда, за по-кратко означаваме с имената на всички менюта през които преминаваме, разделени със знака **/** и завършваме с командата – в случая **File/New Project/Console Application**. В полето **Name** въведете

име на проекта, който ще създадем, например **Hello** и натиснете бутона **OK**.

В резултат средата отваря два нови прозореца (Фиг. 3), с които ще се запознаем по-нататък. Съхраняваме новосъздадения проект с командата **File/Save all**. В диалоговия прозорец на командата може да зададем мястото на проекта върху твърдия диск (текстовата кутия **Location**), както и да променим избраното от средата име на проекта (текстовата кутия **Name**). Освен ако не посочим явно друго име на решението (текстовата кутия **Solution Name**), програмата ще постави там името на проекта. В кутията за избор **Create directory for solution** по премълчаване е поставена отметка, затова програмата създава папка с името на проекта и новосъздаденият проект ще се запази в тази папка със същото име. В тази



Фиг. 3. Текстов редактор на средата и навигацията Solution Explorer

папка ще се съхранят и всички други файлове, които се създават автоматично. Това води до по-добра организация на твърдия диск.

Работни прозорци на средата

За различните си функции средата отваря съответни прозорци. На Фиг. 3 са показани два прозореца, които средата отваря при създаване на конзолно приложение. Вляво на фигурата е прозорецът на средата, в който е отворен текстовият редактор. В него програмистът създава *изходния код* (source code) на програмата. При създаване на нов проект средата поставя там автоматично част от програмата, която е задължителна, за да облекчи работата на програмиста.

При проект, съставен от няколко файла, за всеки от тях се отваря отделна *страница* (tab), надписана с името на файла. На същото място, в отделни страници се изобразяват, когато са необходими на програмиста, и другите ресурси на решението.

В прозореца Solution Explorer (вдясно на Фиг. 3) се показва йерархията от всички ресурси на решението:

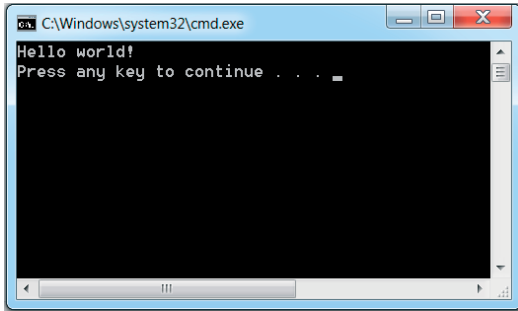
- ❖ Solution 'Hello' е решението, което създаваме. За всяко решение средата създава по един файл с разширение sln (в случая Hello.sln). Файлът, описващ решението, съдържа препратки към проектите в решението. Решението, което сме създали, има 1 проект.
- ❖ След името на проекта се изобразява йерархията от елементи на решението. Отварянето и затварянето на отделните елементи на йерархията става по същия начин както във файловата система. В случая решението има само един проект Hello. За всеки проект средата създава по един файл с разширение csproj, описващ проекта – в случая Hello.csproj Всеки такъв файл може да се разглежда като папка, която съдържа файловете с програмен код или други елементи от проекта.
- ❖ Файлът Program.cs съдържа текста на програмата, написана на езика C#. Този файл е показан в левия прозорец и в него ще дописваме съществената част на нашето приложение.
- ❖ Папките Properties и References съдържат служебна информация за решението, на която няма да се спираме сега.

Работа с компютър

Да довършим работата по създаване на конзолното приложение Hello. Програмата ще прави нещо много просто – ще отвори конзолния прозорец и ще изпише в него текста „Hello world!“ – традиционното

```
static void Main(string[] args)
{
    Console.WriteLine("Hello world!");
}
```

Фиг. 4. Конзолна програма



Фиг. 5. Изпълнение на конзолна програма

поздравление на всеки начинаещ програмист към света. За целта добавете текста `Console.WriteLine("Hello world!");` там където сочи червената стрелка на Фиг. 3, за да получите показаното на Фиг. 4. Програмата е готова и трябва да се преведе на машинен език, т.е да се *компилира*.



В средата Microsoft Visual C# 2010 Express, компилираме с командата `Debug/Build Solution` или с натискане на клавиша `F6`. Компилирайте програмата. Ако компилирането е успешно, то в лентата за състоянието, под двата прозореца на средата, се появява съобщението `Build succeeded`. Командата `Build Solution` извършва и съхраняване на файла. Това не отменя задължението на програмиста да съхранява от време навреме написаното, за да не загуби промените при неочаквани обстоятелства. Можете да разберете кога файлът е бил променен и не е съхранен по това, че средата добъва звезда след името – `Program.cs*`. Стартирайте изпълнимия файл с клавишната комбинация `Ctrl+F5`. Резултатът от изпълнението е показан в конзолния прозорец на Фиг. 5.

Ако програмата съдържа някакви грешки, компилацията няма да завърши успешно. Променете думата `WriteLine` на `Writeline` и се опитайте да компилирате получения текст. Средата ще отвори прозореца `Error List`, в който ще изведе съобщение, описващо допуснатата грешка: `'System.Console' does not contain a definition for 'Writeline'`, т.е. думата `Writeline`, изписана с малко `l` е непозната за компилатора (Фиг. 6). Ако щракнете два пъти върху това съобщение, текстовият показалец на редактора се разполага върху реда, който е причинил грешката, а сбърканата дума се подчертава с червено.

Error List				
1 Error 0 Warnings 0 Messages				
Description	File	Line	Column	Project
1 'System.Console' does not contain a definition for 'Writeline'	Program.cs	12	21	Hello

Фиг. 6. Прозорец за грешки при компилиране

Въпроси и задачи

1. Посочете основни разлики между буквено-цифровия и графичния интерфейс. При кой от двата вида интерфейс създаването на програми е по-трудоемко?
2. Кой от двата вида интерфейс е по-привлекателен за потребителя и защо?
3.  Намерете във файловата система папката на решението `Hello`, в нея намерете папката на проекта `Hello`. В папката на проекта има папка `bin` с две подпапки `Debug` и `Release`. В папката `Release` ще намерите изпълнимата програма `Hello.exe` (Фиг. 7). Стартирайте програмата с двойно щракване върху името ѝ. Какво се получи?
4.  Стартираната в предишното упражнение програма изпълни работата за много кратко време и завърши, което доведе до затваряне на конзолния прозорец и невъзможност да се види резултатът от изпълнението. Затова отворете с програмата `Notepad` текстов файл, въведете в него на първия ред името на изпълнимата програма – `Hello`, а на втория – `pause` и съхранете файла

Речник на българския език, изд. БАН, том I, 1977: Система от правила, които определят последователност от изчислителни операции, прилагането на които води до решението на дадена задача. Алгоритъм на Евклид.

Larousse de la langue Francaise, Lexis, 1979: Съвкупност от правила или предписания за получаване с краен брой операции на определен резултат.

Webster's New Colegiate Dictionary, Webster, 1980: Процедура за решаване на математически проблеми (например намиране на най-голям общ делител) с краен брой стъпки, която често съдържа повтарящи се операции.



Фиг. 1. Ал Хорезми

От цитираните описания на опиятието алгоритъм можем да извлечем някои основни негови характеристики:

1. За създаване на алгоритъм е необходимо множество от *обекти* и *операции* с тях. Например, естествените числа с аритметичните операции и сравняването.

2. Алгоритъмът решава някаква *задача* само с помощта на избраните операции. Например, посочената в едно от описанията – намиране на най-голям общ делител (НОД) на две естествени числа. Задаваните обекти се наричат *входни данни* или *вход*, а получаваните обекти – *резултат* или *изход*. Ако фиксираме входните данни, получаваме *екземпляр* на задачата. Например: „Дадени са числата 12 и 30. Намерете техния НОД.“

3. Алгоритъмът е *процедура*, задаваща последователност от операции, която изпълнена над входните данни на екземпляр на задачата, дава очаквания резултат. Споменатият в едно от определенията *Алгоритъм на Евклид* е такава процедура за намиране НОД на две положителни цели числа. Процедурата трябва да е *детерминирана* – който и да я изпълни над едни и същи входни данни, трябва да получи един и същ резултат.

4. Процедурите, които наричаме алгоритми, трябва да водят до намиране на резултата чрез *краен брой стъпки*, т.е. с прилагане краен брой пъти на допустимите операции. Този брой определя *бързодействието* на алгоритъма. Ако за една задача могат да бъдат построени няколко алгоритъма с различно бързодействие, добре би било да можем да разпознаем най-бързия.

5. В алгоритъма е възможно някои последователности от операции да бъдат повтаряни многократно. Такива повторения се наричат *цикли*. Организирането на цикли е същностна черта на процеса на създаване на алгоритми.

В Урок 4 разгледахме такива процедури за решаване на задачите:

- ❖ да се представи в двоична бройна система число, зададено в десетична система;
- ❖ да се намери сумата на две естествени числа в двоична система;
- ❖ да се сравнят по големина две естествени числа в двоична система;
- ❖ да се преобразува зададено число от осмична в шестнадесетична система.

Представяне на алгоритми

Естествените езици не могат да бъдат използвани за описване на алгоритми, заради възможност от неясно и двусмислено изразяване. Описаният на естествен език алгоритъм може да се окаже недетерминиран. Средства за недвусмислено представяне на алгоритмите са единствено машинните езици и езиците за програмиране, но програмирането не е умение, което се постига лесно. В следващи уроци ще се учим да представяме алгоритмите на езика за програмиране C#.

В този раздел ще се спрем на пол-лесни начини за описване на алгоритми – чрез ограничен естествен език и блок-схеми. На *Фиг. 2a* е представена, на ограничен естествен език, процедура за решаване на линейното уравнение $a \cdot x + b = 0$. Да изпълним процедурата, като зададем за променливата a стойност -7 , а за променливата b – стойност 14 . На стъпка 3, в променливата x се пресмята $-b/a = -14/-7 = 2$ и процедурата ще изведе 2 . Забележете различното значение, което влагаме в знака за равенство при описанието на алгоритмични процедури – пресмята се изразът, който е отдясно на знака и стойността му е новата стой-

ност на променливата отляво. Казваме, че пресметнатата стойност на израза *се присвоява* на променливата.

Ако се опитаме да изпълним тази процедура с коя да е стойност за b и $a = 0$, тогава стъпка 3 няма да може да се изпълни заради невъзможността да се дели на 0. Процедурата не е алгоритмична, защото не завършва успешно за някои стойности на входните данни.

На *Фиг. 2б* към процедурата от *Фиг. 2а* е добавена проверката дали $a = 0$. Тъй като знакът за равенство вече е запазен за присвояването на стойност, *сравняването* на две стойности за съвпадане означаваме с двойно равенство. Сега процедурата няма да спре аварийно при a равно на 0, но пък има друг дефект – ако a е 0 и b е 0, тогава всяка стойност на x е решение, а процедурата ще изведе невярното съобщение "Няма решение". С още една проверка, показана на *Фиг. 2в* като стъпка 6, поправяме и този дефект, за да получим алгоритъма за решаване на линейни уравнения.

Блок-схемният език е използван много в зората на програмирането. Днес той е позагубил своето практическо значение, но остава най-подходящото средство за обучение в техниката на разработване и описване на алгоритми. Основното му качество е, че е графичен и позволява да се представят по-лесно разклоненията, които при линейното представяне с ограничен естествен език или език за програмиране са трудни за проследяване. Всеки блок определя действие, а когато действието е изпълнено, работата на алгоритъма продължава с блока, до който води излизащата стрелка.

Основните типове използвани блокове, представени на *Фиг. 3*, са:

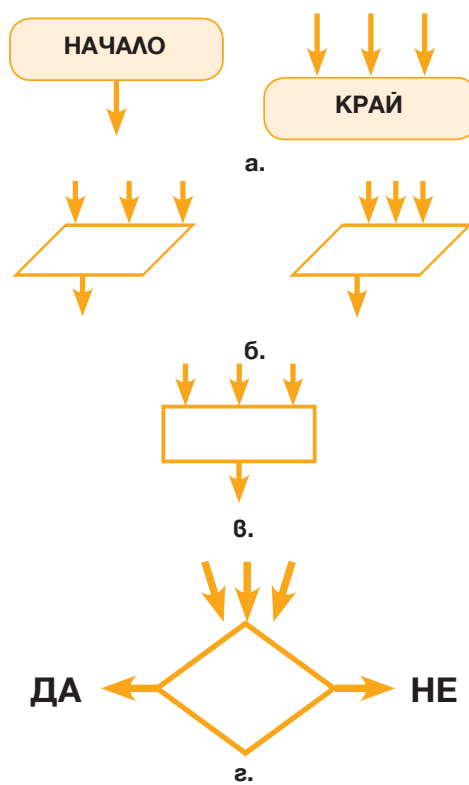
1. Блокове за *начало* и *край* (*Фиг. 3а*). Тези два блока са с овална форма. Блокът НАЧАЛО определя мястото, от което започва изпълнението на алгоритъма и се среща еднократно в блок-схемата. В него не влизат стрелки и излиза една стрелка, показваща кой е следващият блок. Блокът КРАЙ определя място, където се прекратява изпълнението на алгоритъма. Такива блокове могат да бъдат няколко, като трябва да има поне един. От него не излиза стрелка, а може да влизат няколко.

2. Блокове за *вход* и *изход* (*Фиг. 3б*). В първия вид блок се описват входни данни и се указва моментът, в който алгоритъмът получава тези входни данни. Вторият вид блок посочва момента на извеждането и извежданите междинни или крайни резултати. Произволен брой стрелки влизат и точно една стрелка излиза от тези два вида блокове. Алгоритъмът изпълнява предписаният вход или изход и продължава изпълнението с блока, който е посочен от излизащата стрелка.

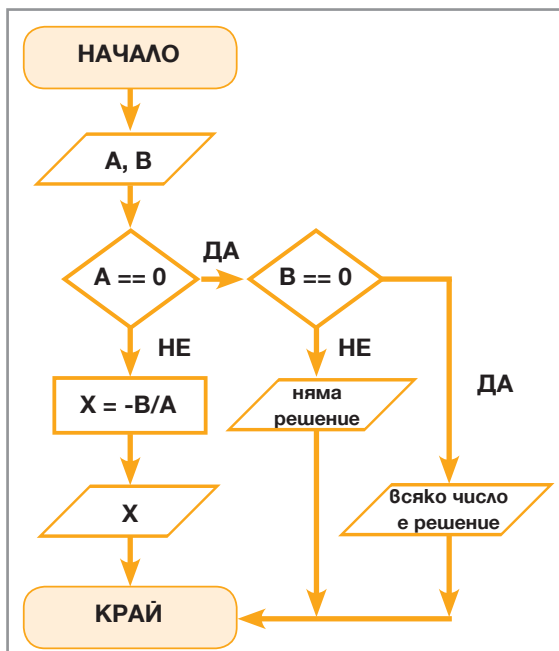
3. *Обработващият* блок (*Фиг. 3в*) има форма на правоъгълник и в него се описват операции над обекти,

1. Въведи a
 2. Въведи b
 3. Пресметни $x = -b/a$
 4. Изведи x
 5. Край
- а.
1. Въведи a
 2. Въведи b
 3. Ако $a==0$ премини_към 6
 4. $x = -b/a$
 5. Изведи x и премини_към 7
 6. Изведи „Няма решение“
 7. Край
- б.
1. Въведи a
 2. Въведи b
 3. Ако $a==0$ премини_към 6
 4. $x = -b/a$
 5. Изведи x и премини_към 8
 6. Ако $b==0$ изведи „Всяко число е решение“ и премини към 8
 7. Изведи „Няма решение“
 8. Край
- в.

Фиг. 2.



Фиг. 3. Блокове



Фиг. 4. Решаване на линейно уравнение

които не са проверки, например изчисляване на изрази и запазване на резултата. Произволен брой стрелки влизат в този блок и точно една стрелка излиза от него.

4. С блока от Фиг. 3г се предписва проверка на някакво условие. В резултат процедурата се разклонява на две в зависимост от това изпълнено ли е условието или не. Блокът има произволен брой входни стрелки и две изходни. Ако условието е в сила, изпълнението продължава с блока, към който сочи стрелката, надписана с ДА, а ако не – с блока, към който сочи стрелката, надписана с НЕ.

На Фиг. 4 е показана блок-схемата на алгоритъма за решаване на линейно уравнение.

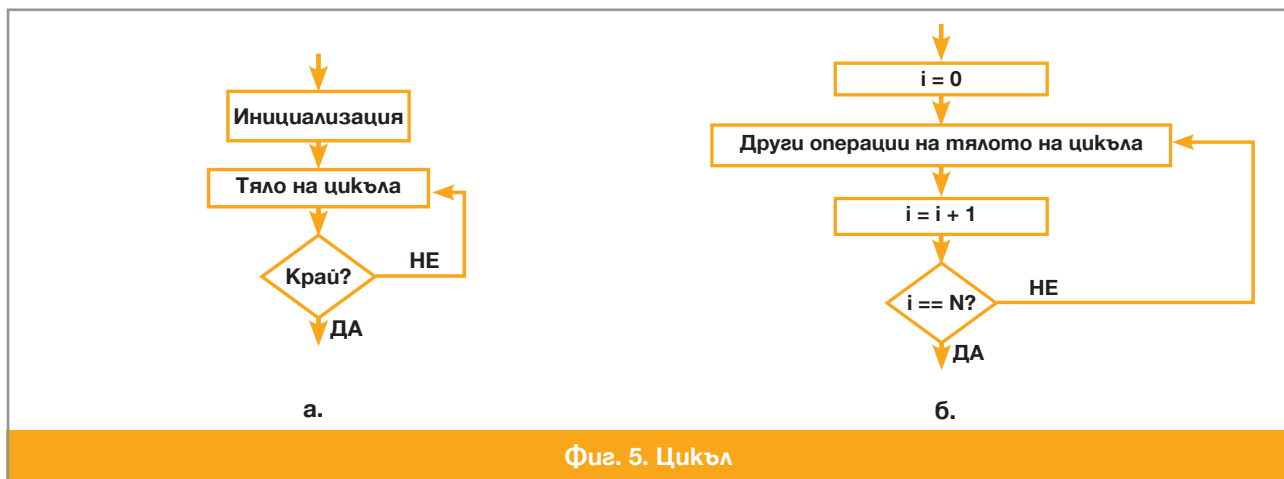
А сега да построим блок-схема за алгоритъма за събиране на числа в позиционна бройна система с основа p .

Ще отбележим две важни особености. Първата е, че индексите в информатиката, както и другите данни, се съхраняват в променливи, чиито стойности могат да се изменят. Следователно, възможно е да записваме операции като $C = a_i + b_i + q$, които в

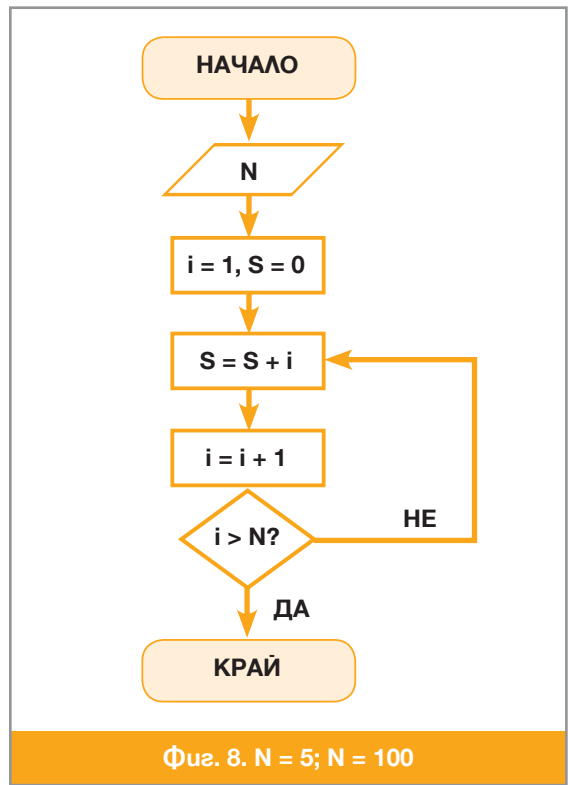
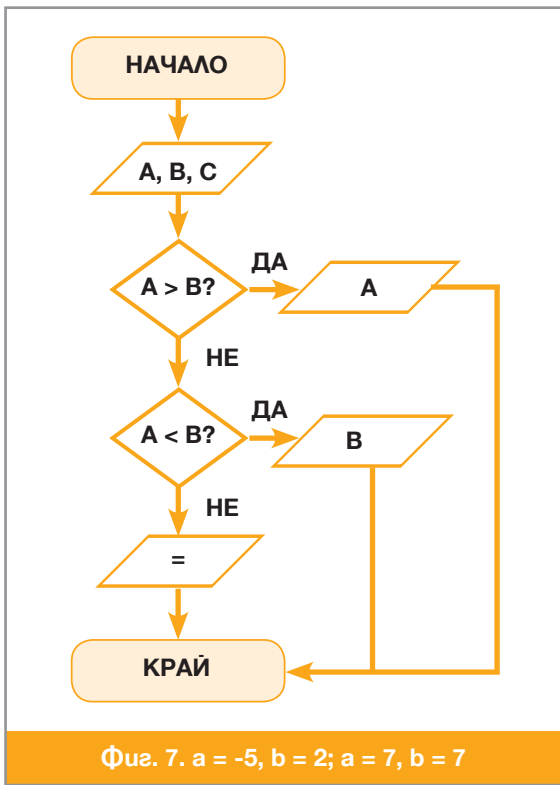
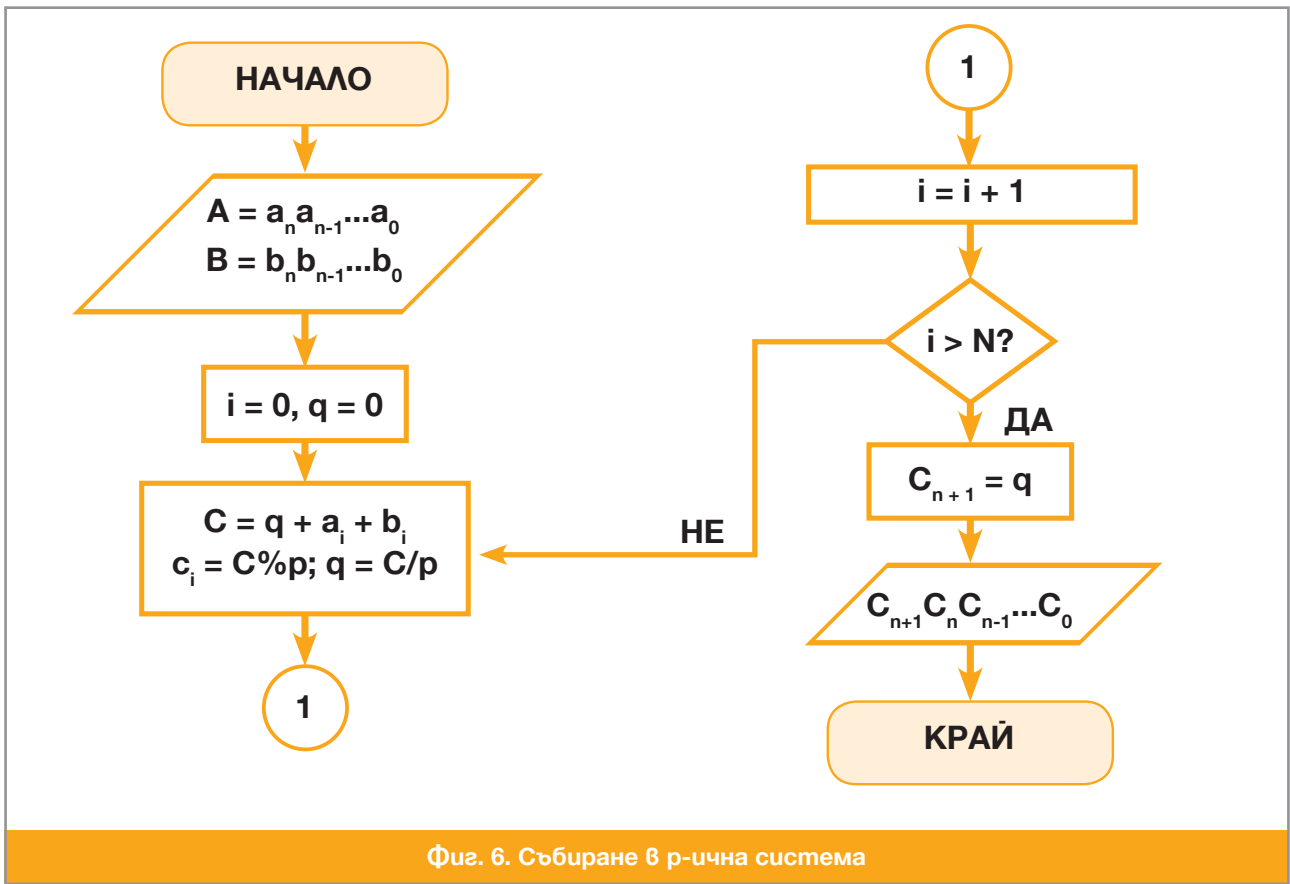
зависимост от стойностите на индекса i имат различно действие. Ако $i = 0$, горната операция пресмята и запомня $C = a_0 + b_0 + q$, но ако $i = 5$, тогава се пресмята и запомня $C = a_5 + b_5 + q$. Това дава възможност при повтаряне на последователности от операции в цикъл, при всяко повторение да извършваме едни и същи действия, но с различни данни, в случая – с различни цифри на двете събираеми и сумата.

Втората особеност е описанието на самия цикъл. Той се осъществява обикновено чрез последователността от блокове, показана на Фиг. 5а. Тялото на цикъла се изпълнява, докато зададеното условие за край не бъде удовлетворено. Често срещан случай е, когато тялото на цикъла трябва да се изпълни определен брой пъти – да речем N . Фрагментът от блок-схемата, осъществяващ такъв цикъл, е показан на Фиг. 5б.

Същественото в тялото на цикъла е изменението на стойността на променливата i , което участва в условието за край. Само така условието може да бъде удовлетворено и да се излезе от цикъла. Още веднъж да обърнем внимание, че **знакът за равенство има тук различен смисъл от този, с който го използваме в математиката**. Математически, означението $i = i + 1$ е еквивалентно на $0 = 1$, което е безсмислица. В блок-схемата $i = i + 1$ означава, че съдържанието на променливата се увеличава с единица. Алгоритъмът за събиране на две числа в p -ична система е показан на Фиг. 6.



Фиг. 5. Цикъл



Въпроси и задачи

1. Обяснете изискването алгоритъмът да е детерминирана и крайна процедура.
2. Защо кулинарните рецепти не са алгоритми?
3. Ако за дадена задача има няколко алгоритъма, какви могат да бъдат критериите, по които да се избере този от тях, който да се използва?
4. Дайте пример на процедура, която удовлетворява всички изисквания за алгоритъм, освен изискването за: а. детерминираност; б. крайност; в. масовост; г. наличие на вход.
5. Прегледайте описанията в Урок 4 алгоритмични процедури. Защо многократното повтаряне на едни и същи, на пръв поглед, стъпки довежда до получаването на смислени резултати?
6. За всяка от блок-схемите на *Фиг. 7* и *Фиг. 8* определете какъв ще бъде изведеният резултат при зададените стойности на входните данни:
7. Съставете блок-схема на процедурата за сравняване на две числа, записани в позиционна бройна система.

Речник

development	дивелъпмънт	развитие; разработване на софтуер
environment	инвайърънтмънт	обкръжение; среда (за разработване на софтуер)
executable	ексик`ютъбл	изпълнима програма
project	пр`ожект	проект
solution	съл`юшън	решение
source (code)	с`урс (код)	текст на програма, изходен код
unsigned	ънсайнд	цяло число без знак

Ръчно програмиране

Първите програмисти – учени и университетски преподаватели в областта на математиката и информатиката – са създавали програми за собствени нужди на **машинен** или **асемблерен език**. Тъй като авторът на програмата е бил и единствен потребител, не се е налагало да се планира предварително нейната функционалност, да се създават документи, описващи вътрешната ѝ логика или функциите ѝ. Проверката за работоспособност на програмата се е извършвала в процеса на използване и когато са се установявали грешки, авторът-потребител е внасял необходимите корекции.

За разработването на програмите не се е използвал софтуерен инструментариум. Не са били изобретени още клавиатурата и мониторът. Програмите са се пишели на хартиени бланки, перфорирани се върху специални носители – *перфоленти* и *перфокарти* и се въвеждали в компютъра от устройства за четене на такива носители. На същите носители се перфорирали и входните данни. Резултатите от работата на програмата се отпечатвали от специализирани механични печатащи устройства върху, потясна или по-широка хартиена лента.

С изобретяването на езиците за програмиране през 60-те години на миналия век, писането на програми става значително по-лесно. Програми създават не само математиците и информатиците, а и специалисти от близки специалности – физици, инженери и др. Възниква професията *програмист*. Програмистите пишат освен програми за свои нужди – компилатори от езиците за програмиране, програмите на операционните системи и т.н. – и програми за нуждите на други потребители.

Технологично програмиране

В началото на 70-те години компютрите получават все по-широко разпространение. Големите компании предпочитат да имат свои компютри за нуждите на управлението на предприятията. Необходимостта от програми става все по-голяма, а програмите все по-сложни. От *ръчната* форма на създаване на програми се преминава към по-съвършената *технологична* форма.

През тези години се налага стилът, наричан *структурно програмиране*. Сложните програми се *описват (специфицират)* като съставени от *модули (подпрограми)*, всяка от подпрограмите също се специфицира и написването на различните подпрограми може да бъде възложено на различни програмисти. Резултат от въвеждането на структурното програмиране е възможността, някои важни подпрограми да бъдат написани веднъж завинаги от много опитни програмисти, да бъдат съхранени в специализирани архиви и да бъдат използвани при създаване на нови програми. Такива подпрограми се наричат *стандартни*, а архивите, в които са съхранени – *библиотеки от стандартни подпрограми*. Така се ускорява програмирането и се гарантира качествено решение на често срещани задачи.

За да могат програмистите да използват подпрограмите, написани от техни колеги, се налагало подпрограмите да бъдат *документирани*. Наложило се и създаването на *потребителска документация* за по-сложните програми, за да може потребителите да работят с тях без помощта на създателите. Така, с програмата се свързват и различни придружаващи я елементи и вместо за програма, вече се говори за *програмен продукт*. Програмните продукти стават *стока*, имат си цена и с тях се търгува, както с всяка друга стока.

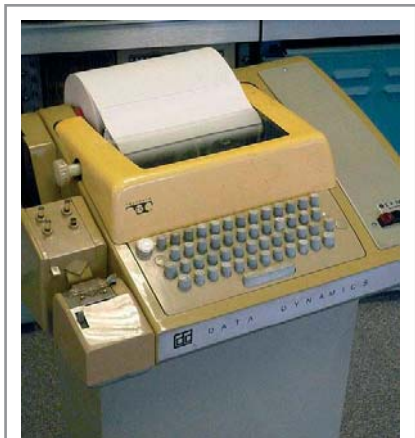
Неизменни етапи от технологичното създаване на програмни продукти са *тестването* и *съпровождането* им. По време на тестването се откриват грешки, дават се предложения относно функционалността, удобствата за потребителя и др. Препоръките, забележките и предложенията се предоставят на програмистите, които извършват необходимите промени. По време на съпровождането на програм-

ния продукт се отстраняват грешки в програмите, неоткрити при тестването. Внасят се необходимите изменения, например когато програмата е свързана с прилагането на нормативен документ, който се променя периодично и др. Затова, много от програмните продукти първоначално се разпространяват в *пробни версии* (алфа версия, бета версия и т.н.) и периодично се *обновяват* (update).

Компютърният терминал

Истинска революция в технологията на програмирането предизвиква създаването на компютърния *терминал* (или *конзола*) състоящ се от:

- ❖ едно входно устройство, от което да се въвежда текстът на програмите и командите към ОС;
- ❖ едно изходно устройство, на което да се извежда това, което работещият на терминала въвежда („ехо“), както и резултатите от изпълнението на програмата.



Фиг. 1. Терминал

Първите компютърни терминали са управлявани от компютъра пишещи машини (Фиг. 1). Разликата между такъв терминал и използваната в живота пишеща машина е, че терминалът работи не с отделни листи, а с непрекъсната хартиена лента.

С изобретяването на терминала се създават принципно нови възможности за облекчаване работата на програмистите. Създават се първите *текстови редактори*. С помощта на текстовия редактор програмистът може да въвежда програмата от терминала във файл, без да се налага да го перфорира на лента или карти. Той може във всеки момент да разпечата избрана част от програмата или цялата програма, за да види състоянието ѝ, след което да зададе нужните му изменения, да съхрани програмата във файла и да стартира компилатора.

Компилаторът също извежда своите съобщения на терминала и програмистът може веднага да се заеме с поправяне на откритите от компилатора грешки. След като програмата се компилира без грешки, програмистът може да я изпълни многократно, задавайки входни данни от терминала или от предварително подготвени с текстовия редактор файлове с данни. Ако до изобретяването на терминала програмистът е можел да разчита на 2-3 компилации и изпълнения на програмата на ден, то при работата от терминал той може да стигне до стотици компилации и изпълнения на програмата за един ден.

При работа с терминал става възможно създаване на програми, улесняващи тестването – *дебъгери*. Дебъгерът позволява програмата да се изпълнява постъпково, като след всяка стъпка се наблюдава състоянието ѝ и така се откриват грешките във функционирането. Създават се и други програми за облекчаване на процеса на програмиране – програми, които подреждат по определени правила текста на програмата и го правят по-лесен за четене, програми които поддържат последователни версии на една програма, за да може бързо да се върнем от неудачна нова версия към по-добрата стара, без да се пазят поотделно двете версии и т.н.

Програми с графичен интерфейс

Постепенно, на мястото на електромеханичния терминал идва електронният – добре познатите днес *клавиатура* и *монитор*. Отначало мониторите са с възможност да изобразяват само знаците на клавиатурната азбука и не носят никаква принципна новост, освен изчезването на неприятния шум на пишещата машина, произвеждан от удара на устройството, носещо съответната буква, по хартията. Затова и създаваните програми са само с буквено-цифров интерфейс. С въвеждането на графичните монитори, и най-вече на цветните графични монитори, се появява възможността за създаване на програми с графичен интерфейс.

Програмите с графичен интерфейс се изграждат по еднотипен начин. При стартирането си такава програма отваря на екрана един *основен прозорец*, в който са изобразени *елементи* на графичния интерфейс *бутони*, *менюта*, *текстови* и *комбинирани текстови кутии*, *поясняващи надписи* и др. С елементи като текстовите кутии и менютата с различни възможности за избор, например, потребителят

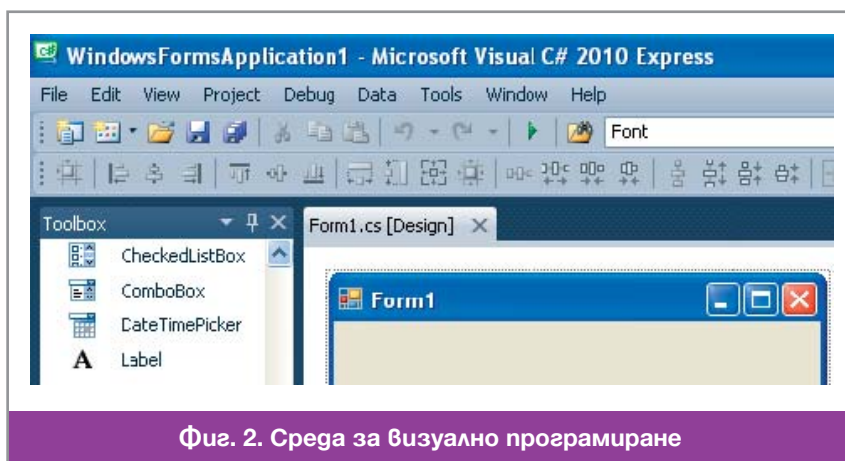
задава данни на програмата, а с бутоните и менютата с команди посочва действията, които програмата трябва да изпълни. По време на работа, за изпълнение на една или друга функция, програмата може да отвори и други – *работни* или *диалогови* – прозорци. Средата за програмиране Microsoft Visual C#, запознаването с която започнахме в предишен урок, както и програмите, които познаваме от уроците по ИТ, са програми с графичен интерфейс.

Визуално програмиране

Създаването на програма с графичен интерфейс може да се осъществи с класически форми на програмиране. Като се използват стандартни подпрограми за създаване на графика да се изчертават прозорците, като за всеки прозорец се помни мястото му на екрана, каква част от него се вижда във всеки момент и т.н. По същият начин се изчертават и отделните елементи на графичния интерфейс и се запомнят техните места в прозореца, съобразявайки се с мястото на прозореца върху екрана, програмират се цветът им на запълване, изписването на съдържащите се в тях текстове и т.н. При това, програмата следи за действията на потребителя и при всяко внесено от него изменение, цялото изображение се прерисува отново. Да се напише такава програма на ръка е изключително трудно. Освен това, какво точно се получава на екрана става ясно след като програмата се компилира и изпълни. Такъв начин на изграждане на програми с графичен интерфейс е изключително неудобен.

Затова, при създаване на приложения програми с графичен интерфейс, се използва по-подходящ стил за програмиране, наричан *визуално програмиране* и съответни инструментални програми. Средата за програмиране Microsoft Visual C#, както показва и името ѝ, е среда за визуално програмиране.

На *Фиг. 2* е показано как изглежда средата, когато се използва за визуално програмиране. За създаване на бъдещите прозорци се използват стандартни графични елементи, наричани *форми*, като на фигурата се вижда една празна такава форма. С настройка на параметри програмистът може да промени размера на формата, запълващия цвят, надписа в заглавната лента и т.н. Върху формата се разполагат другите графични елементи



Фиг. 2. Среда за визуално програмиране

(наричани във визуалното програмиране *контроли* или *компоненти*) като бутони, менюта и др. Компонентата, която е нужна, се избира от инструменталната кутията *Toolbox*, показана вляво на фигурата, и с влачене с мишката се поставя на избраното място във формата. С мишката или чрез настройка на параметрите на съответната контрола, по всяко време може да се променят мястото и/или размерите ѝ.

Благодарение на средствата за визуална разработка, програмистът вижда как ще изглежда интерфейсът още в процеса на проектирането. След като прозорецът е композиран, всичките му атрибути (размер, положение на екрана, свойства, съдържащи се компоненти и т.н.) се записват автоматично като програмен текст в един от файловете на проекта. След това идва ред на класическо програмиране – за написване на подпрограмите, осъществяващи обработката на данните. В езика C# и подобните на него C, C++, Perl и др. подпрограмите се наричат *функции*.

Работа с компютър

Стартирайте средата. Отворете решението, съхранено във файла *Add.sln*, който се намира в папката *Informatika9-10\Razdel_2*. В прозореца *Form1.cs* се намира кодът на програмата, а в прозореца *Form1.cs [Design]* е формата с елементите на графичния интерфейс. Разгледайте ги. Компилирайте програмата и я стартирайте с клавишната комбинация *Ctrl+F5*. Обърнете внимание в кои от трите текстови полета на формата се позволява въвеждането на данни и какъв е резултатът, ако въведете буква вместо цифра.

Въпроси и задачи

1. Защо в първите програмни продукти не се използват бутони, менюта, текстови кутии и др.?
2. В кои от етапите на разработване на програмен продукт може да не участват програмисти?
3. Посочете програмни продукти, които сте ползвали, при които след името им има посочена последната версия.
4. Кога се налагат поправки в програмния продукт, които не са продиктувани от грешки на алгоритъма или програмирането?
5. Намерете в Интернет информация за историята на езика C и направете презентация за създателите му и неговото развитие.

9

Обектно и събитийно програмиране

Представяне на обекти/явления в компютърната памет

Както споменахме в Урок 1, Информатиката се занимава с натрупване на информация за обектите/явленията, съхраняване на събраната информация в компютрите под формата на данни и обработката на съхранените данни. Информацията за един обект, от своя страна, се състои от **множество характеристики** на обекта/явлението и техните стойности. В Урок 4 показвахме как, с представени в двоична бройна система естествени числа, можем да кодираме в компютърната памет всяка интересуваша ни стойност на характеристика и дефинирахме понятието **тип** – множество от стойности, представени еднообразно в компютърната памет и допустимите операции с тези стойности.

Примитивни нарекохме типовете, предоставени от хардуера посредством езика за програмиране. Всеки примитивен тип се отличава с *дължината* на полето от паметта което използва и начина за представяне на данните в полето и си има уникално за езика име. Езикът за програмиране C# предоставя следните примитивни типове, групирани по начина на представяне:

- ❖ *Целочислените* типове са 8, по два за всяка от дължините 1, 2, 4 и 8 байта – един със знак и един без знак, съответно: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong`;
- ❖ *Дробните* типове с плаваща запетая в двоична система са два, с дължина 4 и 8 байта съответно: `float` и `double`;
- ❖ *Дробен* тип с плаваща запетая в десетична система: `decimal`;
- ❖ *Знаковият* тип `char` съвпада с беззнаковия целочислен `ushort`, но на стойностите му винаги се гледа не като на числа от 0 до 65535, а като на *знаци* от таблицата Unicode;
- ❖ *Булевият* тип – `bool` – се използва в програмирането за означаване на резултата от проверка на условие и има две стойности `true` (истина, условието е изпълнено) и `false` (неистина, условието не е изпълнено).

Езикът C# предоставя и два типа, които не са примитивни, но е удобно да бъдат разглеждани като примитивни. Стойностите на типа `string` – *низ от знаци* (или просто *низ*) – са всевъзможните последователности от знаци, т.е. последователности от стойности на типа `char`. Естествено е, че такъв тип няма собствена дължина – всяка конкретна стойност на типа е със специфична дължина. С втория непримитивен тип, за който е удобно да го разглеждаме като примитивен е типът `object`. С него няма да се занимаваме тук.



Работа с компютър

Стартирайте средата. Отворете приложението Hello, което направихме в Урок 6, или от файла Hello.sln, който се намира в папката Informatika9-10\Razdel_1. Вместо реда

```
Console.WriteLine("Hello world!");
```

напишете реда

```
Console.WriteLine(sizeof(bool));
```

Съхранете получената програма под името Sizes, компилирайте я и я изпълнете. В конзолата програмата ще изведе стойност 1, което означава, че размерът на (англ. size of) типа bool е 1 байт. Вместо реда

```
Console.WriteLine(sizeof(bool));
```

напишете реда

```
Console.WriteLine(sizeof(string));
```

и се опитайте да компилирате получената програма. Защо програмата не може да определи размер за стойностите на типа string?

Определете по посочения начин размерите на останалите примитивни типове на езика C#.

Обекти и обектно-ориентирано програмиране

Броят на примитивните типове обикновено е доста ограничен. За представяне на сложни обекти (процеси, явления) езикът за програмиране предоставя механизъм за дефиниране на *потребителски типове*. Съществуват различни такива механизми, но господстващ в момента е механизъмът наричан *клас от обекти*, а процесът на създаване на програми с използването на класове от обекти – *обектно-ориентирано програмиране* (ООП). Ще се запознаем с основните елементи, които изграждат този механизъм.

Програмният *обект* е това, което в програмата съответства на реалния обект или явление. Например, за програма, която е предназначена да обслужва учебния процес, естествено е да дефинираме обекта Ученик. Основание за това е фактът, че за всеки ученик трябва да бъдат запазени в компютъра няколко характеристики, наричани *атрибути* (или *свойства*) на обекта: **име** и **дата на раждане** – стойности от типа string; **идентификатор на класа**, в който ученикът учи – също стойност от типа string; **номер в класа** – цяла стойност без знак от типа byte, например, тъй като броят на учениците в клас не е повече от 20-30; **оценки по учебните предмети**; **брой отсъствия** и др.

С всеки обект в компютърната програма обикновено се свързват някакви обработки, извършвани в компютъра от подпрограми. Подпрограмите, свързани с обработката на определен обект се наричат *методи*. Например, за обекта Ученик може да има методи, които определят името му, датата му на раждане, средния му успех или кой да е друг атрибут и т.н.

Всички еднотипни обекти – с еднакви атрибути и методи – образуват *клас от обекти* или просто *клас*. Всеки конкретен обект със специфични стойности на атрибутите се нарича *екземпляр* на класа. Класът е средството на езика за обединяване на всички **еднотипни обекти** и е аналог на това, което извън ООП нарекохме потребителски тип. Както ще видим по-нататък, всеки от примитивните типове в C# е оформен и като клас. В езика C# класът се обозначава с ключовата дума class. Текстовият редактор на средата оцветява имената на класовете в зелено.

Подобно на стандартните подпрограми в процедурното програмиране, езиците за ООП предоставят на програмиста *библиотеки от стандартни класове от обекти*. Това са често използвани класове с общоприети методи или класове, за които трудно бихме написали сами съответни методи. В програмата Hello, която създадохме в Урок 6, използвахме стандартно дефинирания в C# клас Console. Този клас има един единствен обект, носещ името на класа – конзолният прозорец, свързан с програмата. В програмата използвахме метода WriteLine(<аргумент>) на класа Console, който извежда в конзолния прозорец, от мястото, на което се намира показалецът, зададения му в скоби аргумент и след това премества показалеца на конзолата на следващ ред.

Когато искаме да *приложим* метод на клас от обекти върху избран обект от класа, използваме оператора за извикване на метод: **<име на обекта>.<име на метода>(<аргументи>);**
например: `Console.WriteLine("Hello world!");`

Работа с компютър

Отворете отново програмата Hello. Под реда

```
Console.WriteLine("Hello world!");
```

започнете да пишете нов ред:

```
Console.
```

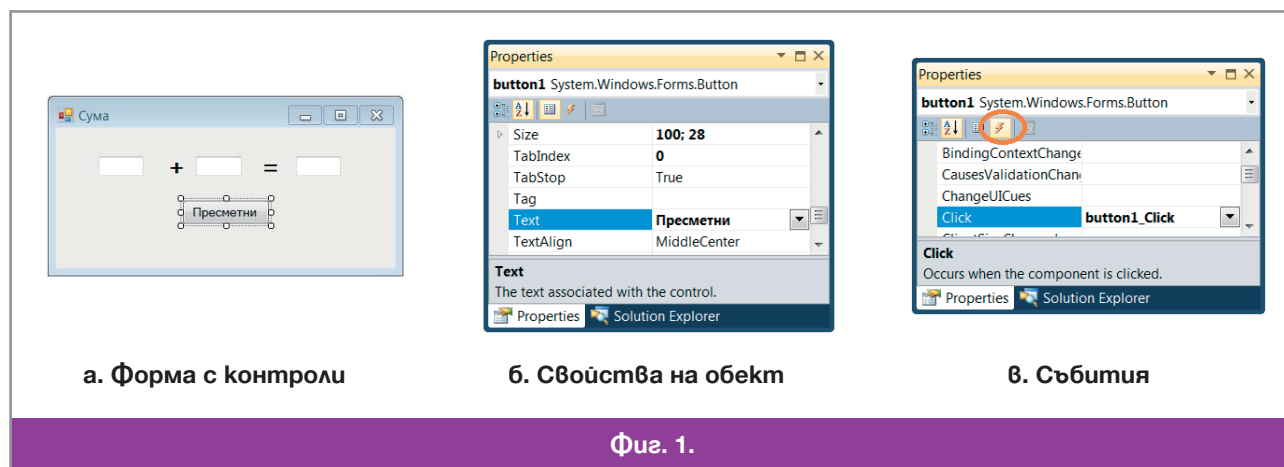
Когато изпишете първите няколко букви от името на класа, средата ще ви предложи падащо меню, от което да изберете нужния обект. По същия начин, когато изпишете разделящата точка, средата ще отвори падащо меню с всички приложими методи. Изберете метода `Write` и го приложете два пъти:

```
Console.Write("Hello");  
Console.Write(" world!");
```

Компилирайте програмата и я изпълнете. Обяснете каква е според вас разликата между методите `Write` и `WriteLine`.

Обекти и визуално програмиране. Събития. Изключения

ООП е много удобен подход за целите на визуалното програмиране, затова всяка среда за визуално програмиране предоставя библиотека от стандартни класове от обекти за елементите на графичния интерфейс – екранните форми и различните *компоненти*. В програмата, която разгледахме в Урок 8, е използван един обект от класа `Form` – екранната форма, върху която са поставени компонентите: три текстови кутии – обекти от класа `TextBox`, два етикета – обекти от класа `Label` и един бутон – обект от класа `Button`. Формата може да разгледате в прозореца `Form1.cs[Design]` (Фиг. 1а). Всички свойства на един графичен елемент са събрани в прозорец, който се отваря като щракнем с мишката върху компонентата и след това върху бутона `Properties` (🔗) в менюто с инструменти. Част от свойствата на обекта `button1` от класа `Button` са показани на Фиг. 1б. На фигурата се виждат, например, размерите на бутона (свойството `Size`) – ширина 100 и височина 28, както и надписващият текст `Пресметни` (свойството `Text`).



Много характерно за класовете на компонентите е понятието *събитие* (event). Щракване с левия бутон на мишката върху графична компонента и редица други действия на потребителя предизвикват събития. Програмата, за която се отнася събитието, получава известие за случилото се и трябва да реагира така, както е предвидил програмистът. Тази реакция на програмата наричаме *обработка на събитията*. За да се види какви събития са валидни за един елемент на графичния интерфейс, трябва да се отвори прозорецът му `Properties` и в менюто да се натисне бутонът `Events` (🔗).

На Фиг. 1в са показани част от събитията, свързани с бутона `button1`. Маркирано в синьо е събитието `Click`, което възниква при щракване върху компонентата с левия бутон на мишката. Както се вижда от фигурата, при настъпване на събитието ще бъде изпълнена функцията `button1_Click`. Затова в тази функция програмистът трябва да напише предвидената обработка – в случая, събирането на числата от левите две текстови полета и записване на резултата в дясното текстово поле.

Специфичен вид събития, свързани с компонентите, са *изключенията* (англ. *exception*). По време на работа с програмата потребителят може да допусне грешка. Например, да въведе в някое от текстовите полета за събираемите вместо цифри, други знаци или да натисне бутона `Пресметни`, когато някое от полетата за събираемите е празно. В двата случая пресмятането на сумата няма да е възможно и програмата трябва да реагира с отказ да извърши пресмятането, вместо да покаже някакъв безсмислен резултат. Например, да отвори прозорец с предупреждаващо съобщение в първия случай или да приеме, че съдържанието на празното поле е нула, във втория случай.

Програмният код, който реагира на допуснатите грешки наричаме *обработка на изключенията*. Програмистът трябва да предвиди колкото се може повече от възможните грешки и неговата програма да реагира на всяка една от тях със съответна обработка.

Въпроси и задачи

1. Каква е разликата между свойство на елемент на графичния интерфейс и събитие, свързано с този елемент?
2. За текстовите кутии програмистът има възможност да забранява писането, като постави стойност `False` срещу надписа `ReadOnly` (само за четене). Какво е `ReadOnly` – свойство или събитие?
3. Колектив от програмисти, в който участвате, трябва да направи програма, която е електронен дневник на класа. На вас се пада да разработите формата, в която се въвеждат срочните оценки на един ученик. Какви компоненти бихте поставили във формата и какви проверки за грешно въвеждане бихте направили за тези от тях, в които се въвеждат данни?
4. За програма, свързана с информационното обслужване на учебния процес, трябва да се разработят класовете от обекти „Училище“, „Клас“, „Паралелка“ и „Ученик“. За всеки от класовете предложете подходящи атрибути и методи.
5. В текстово поле на прозорец на програма трябва да се въведе ЕГН на човек. Кои са възможните грешки, за които програмистът трябва да направи проверка?

10

Програма на C# - основни елементи

Структура на програмата на C#

Програмите на езика C# са съставени от подпрограми, наричани *функции* и *класове от обекти*.

Всяка дефиниция на функция в езика C# има следния синтаксис:

```
<тип_на_функцията> <име_на_функцията> (<списък_от_аргументи>)  
{ <тяло_на_функцията> }
```

Например,

```
static void Main(string[] args)  
{ Console.WriteLine("Hello world!"); }
```

Тип на функцията е типът на стойността, която тя пресмята. Не всяка функция пресмята някаква стойност. Типът на функциите, които не пресмятат стойности, се означава с `void`. Към типа на функцията могат да се добавят различни модификатори – `static`, `private` и др., с предназначението на които няма да се занимаваме.

Функцията се идентифицира с *името си*, *броя* и *типа на аргументите* в *списъка от аргументи*. Списъкът от аргументи на една функция може да е празен, но ако не е, е последователност от двойки „тип – име на променлива“. Имената на функции, както и другите имена в езика се съставят по правила, с които ще се запознаем в този урок. В *тялото на функцията* са описани *данните*, които функцията обработва, ако има такива и *операторите* на езика за програмиране, които реализират възложената на програмата обработка.

Всяка програма задължително има една поне една функция с име `Main`. Това е функцията, която е *входна точка* на програмата, т.е. когато стартираме програмата, изпълнението ѝ започва от функцията с име `Main`. **Всяка функция трябва да бъде част от някакъв клас.** Затова в нашия пример от Урок 6 функцията `Main` е включена в клас, наречен `Program`.

Ключови думи

Езикът C# използва едно множество от думи за свои вътрешни цели. Тези думи наричаме *служебни* или *ключови*. Всяка ключова дума има строго определено предназначение в езика, което не може да се променя.

Ключови думи на езика C# са имената на типовете – `int`, `uint`, `float`, `char`, `bool` и т.н., включително думата `void`, означаваща отсъствие на определен тип. Ключова е думата `class`, с която конструираме нови класове. Ключови са и думите `true` и `false`, с които означаваме двете възможни стойности на типа `bool`.

Ключовите думи на езика C# са 77 и са подредени лексикографски в следната таблица.

<code>abstract</code>	<code>as</code>	<code>base</code>	<code>bool</code>	<code>break</code>
<code>byte</code>	<code>case</code>	<code>catch</code>	<code>char</code>	<code>checked</code>
<code>class</code>	<code>const</code>	<code>continue</code>	<code>decimal</code>	<code>default</code>
<code>delegate</code>	<code>do</code>	<code>double</code>	<code>else</code>	<code>enum</code>
<code>event</code>	<code>explicit</code>	<code>extern</code>	<code>false</code>	<code>finally</code>
<code>fixed</code>	<code>float</code>	<code>for</code>	<code>foreach</code>	<code>goto</code>
<code>if</code>	<code>implicit</code>	<code>in</code>	<code>in (genetic)</code>	<code>int</code>
<code>interface</code>	<code>internal</code>	<code>is</code>	<code>lock</code>	<code>long</code>
<code>namespace</code>	<code>new</code>	<code>null</code>	<code>object</code>	<code>operator</code>
<code>out</code>	<code>out (genetic)</code>	<code>override</code>	<code>params</code>	<code>private</code>
<code>protected</code>	<code>public</code>	<code>readonly</code>	<code>ref</code>	<code>return</code>
<code>sbyte</code>	<code>sealed</code>	<code>short</code>	<code>sizeof</code>	<code>stackalloc</code>
<code>static</code>	<code>string</code>	<code>struct</code>	<code>switch</code>	<code>this</code>
<code>throw</code>	<code>true</code>	<code>try</code>	<code>typeof</code>	<code>uint</code>
<code>ulong</code>	<code>unchecked</code>	<code>unsafe</code>	<code>ushort</code>	<code>using</code>
<code>virtual</code>	<code>void</code>	<code>volatile</code>	<code>white</code>	

Разгледайте съдържанието на файла `Program.cs` в програмата `Hello` от Урок 6 и отбележете ключовите думи в него. В какъв цвят текстовият редактор на средата оцветява ключовите думи?

Имена (идентификатори)

Вече знаем, че функциите и методите на програмата се идентифицират с имената си. С имената си се идентифицират и *променливите* – полетата в оперативната памет, в които се съхраняват необходимите на програмата данни, както и класовете. *Имената* (или *идентификаторите*) в C#, както и в повечето езици за програмиране, са низове от латински букви, цифри и знакът за подчертаване (`_`), започващи с латинска буква или знака за подчертаване. **За имена не могат да се използват ключови думи.** Например, `asd43`, `asd_43`, `a_s_d_43`, `_asd43` и `asd43_` са допустими имена в C#, а `asd-43`, `43asd` и `int` – не са допустими.

В езика C# малката буква и съответната ѝ главна се считат различни. Затова `asd43`, `Asd43` и `ASD43` са различни имена. Добре е да се съобразяваме и със следните препоръки при избор на имена, които са свързани с конвенцията CLS за съвместимост между различните езици, използвани в средите на Microsoft:

- ❖ не използвайте знака за подчертаване за начало на името – създателите на компилатори използват този знак за специфични цели на компилацията и свързването на компилираните модули в програми;
- ❖ не създавайте имена, които се различават едно от друго само по това, че в едното е използвана малка, а в другото съответната главна буква (например, `name` и `Name`);
- ❖ избирайте имената така, че да подсещат за предназначението на именуваното;
- ❖ започвайте името с малка буква;
- ❖ ако името е съчетание от няколко думи, то втората и всяка следваща дума да започват с главна буква (например `userName`).

При създаване на дълъг програмен код се използват много имена на променливи, функции, методи и класове, което е предпоставка за възникване на конфликти на имена и може да направи кода неясен. *Пространство от имена* е средството на езика C# за решаването на подобни проблеми.

Синтаксисът на директивата за обявяване на пространство от имена, т.е. правилото по което тази директива се изписва, е:

```
namespace <име_на_пространството_от_имена>
{ <програмен код, включен в пространството от имена> }
```

Две еднакви имена, деклариращи в различни пространства от имена, се считат различни. Средата Visual Studio следва препоръката – всеки клас да бъде включен в някое пространство от имена. Затова, в нашия пример от Урок 6, единственият клас на програмата – класът **Program**, съдържащ нейния код, беше включен в незадължителното пространство от имена **Hello**.

За да може да се използва от програмиста програмният код, включен в някое пространство от имена, съответното пространство трябва да бъде включено в програмата с помощта на директивата **using**.

Директивата за включване на пространство от имена:

```
using <име_на_пространството_от_имена>;
```

прави достъпни за използване в програмата всички имена, деклариращи в посоченото като аргумент пространство от имена. Ако в една програма трябва да бъдат използвани две еднакви имена от различни пространства, тогава пред всяко от тях трябва да се постави името на пространството, последвано от точка. Същото правило се прилага и когато пространството от имена не е включено в програмата.

В нашия пример от Урок 6, стандартният клас **Console** е дефиниран в служебното пространство от имена на средата **System**. Затова пространството **System** е включено в програмата с директивата **using System**; В противен случай, извикването на метода **WriteLine** трябва да изглежда така:

```
System.Console.WriteLine("Hello world");
```

В автоматично генерирания код са включени и други пространства от имена, които не са необходими за тази проста програма. Изтрийте съответните редове и компилирайте отново програмата.

Коментари

Коментарът е текст в програмния код, който не е същинска част от програмата и не се компилира. Използва се за описване на ролята на променливите, за обяснения на алгоритъма като цяло или на някои от използваните оператори. Понякога част от кода се коментира, за да не се компилира. По този начин може да се тества само некоментираният код, за да се локализира местата на грешките.

Има два вида коментари:

- ❖ коментар на един ред – започва с две наклонени черти (//) и продължава до края на реда;
- ❖ коментар на повече от един ред – започва със знаците /* и завършва с */.

Оформяне на програмата

За компилатора е абсолютно безразлично как ще бъде подреден текстът на програмата, стига да е написан по правилата на езика за програмиране. При въвеждане и редактиране на кода на програмата, обаче, е добре да се спазват няколко общоприети правила, за да стане програмата по-лесна за четене, тестване и внасяне на изменения:

- ❖ ако нямате нещо друго предвид, изписвайте всеки оператор на отделен ред, т.е. разполагайте програмата „на дължина“, а не „на ширина“;
- ❖ изписвайте всяка затваряща скоба } със същото отместване от началото на реда, с което е изписана и съответната ѝ отваряща скоба {;
- ❖ използвайте коментари, за да може програмният код да бъде разбраем, както за вас след известно време, така и за други програмисти.

Работа с компютър

От казаното в този урок става ясно, че програмата **Hello**, използвана в Урок 6 като пример, може да бъде опростена доста и, след като ѝ добавим коментари, да добие вида:

```

class Program //програмата трябва да е в клас
{
    /* всяка програма трябва да съдържа
    главна функция с име Main */
    static void Main(string[] args)
    {
        System.Console.WriteLine("Hello world!");
    }
}

```

което е минималната форма на конзолно приложение. Редактирайте програмата Hello по указания по-горе начин, компилирайте я и я изпълнете, за да се убедите в работоспособността ѝ.

Въпроси и задачи

1. Въведете таблицата с ключовите думи в текстов документ. Срещу всяка ключова дума, която сме споменали, отбележете предназначението ѝ. Продължавайте да отбелязвате предназначението на всяка ключова дума, щом споменем за нея в поредния урок.
2. Може ли като име на променлива да се използва ключова дума? Защо?
3. Коя, според вас, е причината, да се препоръчва имената да напомнят предназначението им в програмата?
4. Кой от следните низове е правилно име в C#: `Abc`, `клас`, `klas`, `1klas`, `byte`, `22`, `s22`? За всеки низ, който не може да е име, посочете правилото, което е нарушено.

11

Променливи и константи. Инициализация

Деклариране на променливи

Данните, необходими за работата на една програма – както входните, така и тези, които се пресмятат по време на работата на програмата, се съхраняват в *променливи*. Всяка променлива е от някакъв тип и заема съответното количество байтове в паметта. За да може една променлива да се използва в програмата, тя трябва да бъде декларирана:

Операторът за деклариране на променлива има следния синтаксис:

```
<тип> <име на променлива>; .
```

Например,

```
int x; double pi; string nameAndFamily; .
```

Няколко променливи от един и същ тип могат да се декларират с един оператор:

```
int x, y, z; .
```

Забележете, че операторът за деклариране на променливи трябва непременно да завършва със знака точка и запетая!

Декларирането на променлива означава, че в оперативна памет се определя поле с размера, необходим за съхраняване на стойности от посочения в оператора тип. Например, за декларираната променлива `x` от типа `int` ще бъде заделено поле от паметта с размер 4 байта, тъй като това е размерът за този тип.

Както и в математиката, стойността на една променлива може да е различна във всеки един момент от работата на програмата, откъдето е и названието променлива. Затова често говорим не просто за стойността на променливата, а за нейната *текуща стойност*.

Когато някъде в програмата поставим **името на променлива**, това означава, че на това място искаме да използваме нейната **текуща стойност**.

Всяка променлива от примитивен тип може да получава стойност от множеството от допустимите стойности на типа. Всяка конкретна стойност на типа се нарича *константа*. Всеки език за програмиране има правила за изписване на константите.

Константи за **целочислените типове** са целите числа, които в езика C# както и в математиката, изписваме в десетична бройна система – с последователност от десетични цифри. Например 123, +2048, 007, -1734. Поставянето на водещи нули е допустимо и не променя стойността на числото, както и знакът +. Знакът минус може да се добавя в началото само за константи от типовете със знак – sbyte, short, int и long.

Константи за **дробните типове** float и double са дробните числа, с или без знак, като десетичният знак в C# е **точка**. Например, 123.2048, 0.07, -1.734. За дробните константи е допустимо да се представят и в експоненциален вид – с мантиса и десетичен порядък, при това за мантисата не е нужно да се грижим да е подравнена по начина, по който ще се представи в паметта. Мантисата и порядъкът се разделят с голямата латинска буква E. Например, дробната константа 3.14 може да се представи в програма на C# като 314E-2 или като 0.314E1.

Константните стойности за **булевия тип**, както вече споменахме са true (истина) и false (неистина). Константните стойности на **знаковия тип** char са знаците от таблицата Unicode. Тази таблица съдържа 65536 позиции и в нея са включени всички знаци от клавиатурата, малките и главните букви на латиницата и кирилицата, буквите на всевъзможни други използвани по света азбуки, както и много други знаци. Част от позициите в таблицата са свободни за бъдещо използване. Всеки знак от тази таблица има *код* – число от 0 до 65535, което се побира в 2 байта.

Съответствието между знака и неговия код, задавано от таблицата Unicode е фактически стандарт за всички производители на компютри и софтуер. Това унифицира използването на знаците по целия свят. При по-малки кодови таблици както еднобайтовата ASCII, текстово съобщение написано на един компютър, може да изглежда по съвсем друг начин, визуализирано на друг компютър. Константите от знаков тип се изписват като съответния знак, когато го имаме на клавиатурата, се постави в апострофи. Например: 'a', 'Z', '7', '*', '@', ' ' и т.н. За знаците, които не са изписани на клавиатурата в кавичките пишем обратна наклонена черта (\) и кода им – '\1024'.

Константните **низове** (типа string) са последователност от знаци от таблицата Unicode, поставени в кавички. Например, "hello", "Hello", "HELLO WORLD", "My name is: ", "=" и т.н. Малките и главните букви имат различни кодове, т.е. са различими (низът "hello" е различен от низа "Hello").

Инициализиране на променливи

Преди да използваме една променлива в езика C#, трябва да я *инициализираме*, т.е. да ѝ зададем начална стойност. Това може да стане или по време на декларирането или в *оператор за присвояване*, преди променливата да бъде използвана. Ако това не е направено, компилаторът ще изведе съобщението за грешка Use of unassigned local variable <име_на_променливата> (Фиг. 1).

Инициализирането на променливи може да стане в оператора за деклариране

```
<тип> <име_на_променлива> = <константа> ; ,
```

например

```
int x=0, y=-1, z=3;
```

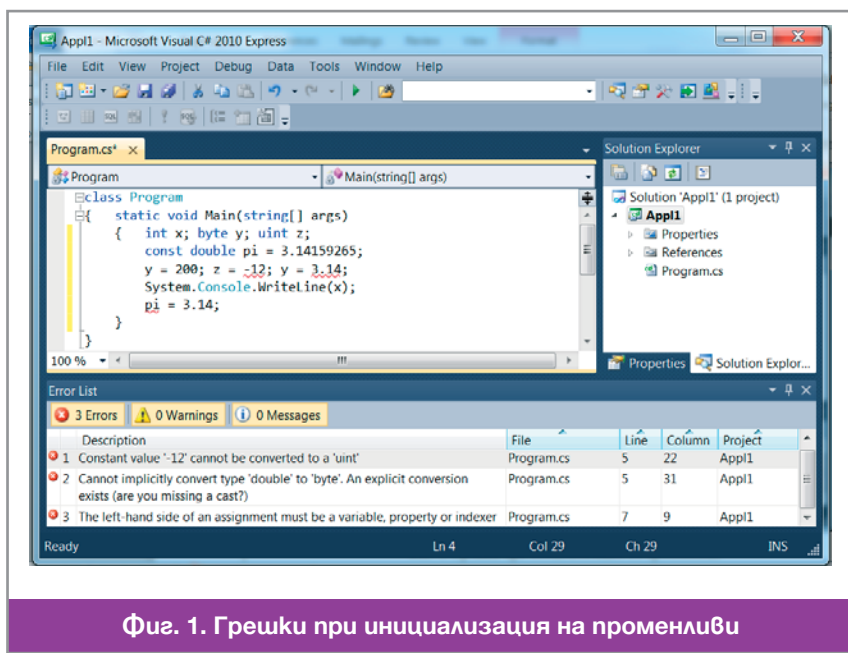
а може да стане и по-късно с *оператора за присвояване на стойност*:

```
<име_на_променлива> = <константа>;
```

Например

```
int x, y, z; ... x=0; y=-1; z=3; .
```

Обърнете внимание на нещо, което вече споменахме в урока за алгоритми. Ролята на знака за равенство в информатиката е различна от ролята му в математиката. Затова и при четенето на записа тук постъпваме по различен начин. Ако в математиката записът $x = 12$ е означение на факта „x е равно на 12“, то в програмирането записът `x = 12;` е означение на действието „на променливата x се присвоява стойност 12“.



Фиг. 1. Грешки при инициализация на променливи

Освен това, многократното изписване на дълга константа отнема повече време. Най-важната причина е, че ако след време решим да променим стойността на константата, ще бъде изключително трудно да намерим всички места в програмата където тя се среща. Ако пък в програмата се използват две различни по предназначение константи, които в един момент случайно имат еднаква стойност, тогава на всяка цена трябва да се оформят като именуван константи, защото в противен случай програмата става трудна за четене.

За създаване на **именувана константа** използваме оператора:

```
const <тип> <име на константата> = <константа>;
```

Например, `const double pi = 3.14159265;`

Именуваната константа може да бъде използвана в програмата само там, където може да се използва константа. Не е възможно да се промени стойността на именуваната константа и не може да ѝ се даде като стойност, стойността на променлива.

Работа с компютър

Напишете следното конзолно приложение:

```

class Program
{
    static void Main(string[] args)
    {
        int x; byte y; uint z;
        z = -12; y = 3.14;
        System.Console.WriteLine(x);
    }
}

```

и го съхранете под името App1. В него сме използвали възможността на метода WriteLine на класа Console да извежда стойността на аргумент от числов тип. Текстовият редактор ще покаже в прозореца за грешки неправилните инициализации – *синтактически ориентиран редактор*, т.е. проверяващ синтаксиса на операторите преди още да е поискано компилиране. Внесете и други възможни грешки при изписването на числовите константи, например да поставяте букви в тях. Проверете реакцията на редактора на неправилно изписване на знакова константа и низ – например, липсващ апостроф или кавичка.

Добавете в програмата декларацията за именуван константа със стойност първите няколко цифри след десетичната точка на числото π . Опитайте се да промените стойността на именуваната константа, след като вече е декларирана и вижте реакцията на редактора (Фиг. 1). Опитайте се да декларирате именуван константа, като вземете за нейна стойност тази на променлива и вижте реакцията на редактора.

Поправете всички допуснати грешки, компилирайте програмата и я изпълнете.

Въпроси и задачи

1. Кои от следните декларации на променливи са правилни:
а) `int a;` б) `int byte x;` в) `string 1s;` г) `byte 'a', b, c.`
2. Кои от следните низове са правилни константи от типа `uint`:
а) `-1524;` б) `123E4;` в) `000;` г) `1000000000000;` д) `1.`
3. Кои от следните низове са правилни константи от типа `double`:
а) `-1524;` б) `123E4;` в) `000.;` г) `.1000000000000;` д) `1.1.`
Упътване. Ако не можете да прецените сами, поставете съответната константа на подходящо място в програмата `App11`.
4. Кои от следните низове са правилни константи от типа `string`:
а) `"-1524";` б) `'1''2''3';` в) `abcde;` г) `"1000000000000 ;` д) `" "` .

12

13

Операции и изрази

Операции, операнди, знаци на операциите

Както споменахме вече в предишен урок, за всеки тип данни са присъщи и съответни *операции*. Всяка операция се прилага върху един или няколко *операнда* (или *аргумента*) и връща като *резултат* някаква стойност. В зависимост от броя на операндите си, операциите се делят на такива *с един операнд* (или *едноаргументни*), *с два операнда* (или *двухаргументни*), *с три операнда* (или *триаргументни*) и т.н.

Обичайно е за всяка операция да имаме *знак на операцията* и да разполагаме аргументите около знака. Така например, за едноаргументните операции имаме две възможности за поставяне на знака:

- ❖ *префиксен* запис на операцията: `<знак_на_операцията> <операнд>;`
- ❖ *постфиксен* запис на операцията: `<операнд> <знак_на_операцията>.`

Едноаргументна операция, например, е операцията за обръщане на положително число в отрицателно, с поставяне на знака минус пред числото. Префиксен или постфиксен е този запис? Тъй като знаците на клавиатурата, които можем да използваме за знаци на операции не са много, в някои случаи за означаване на операции се използват комбинации от няколко знака.

Едноаргументната операция може да бъде записана и по други начини – *функционален запис*. Такава едноаргументна операция е, например, познатата ни операция за определяне броя на байтовете, които заема стойност от някакъв тип в оперативната памет – `sizeof(<тип>)`. При функционалния запис аргументът, в случая някой от възможните типове, се поставя в кръгли скоби след името на операцията, в случая – `sizeof`.

За двухаргументните операции, каквито са повечето от операциите в математиката, общоприет е *инфиксният запис*, при който знакът на операцията се поставя между двата операнда: `<операнд> <знак_на_операцията> <операнд>`. Пример за двухаргументни операции са събирането, изваждането, умножението и деленето.

Операции в C#

Езикът C#, както и езикът C, от който той произхожда, предлага голямо разнообразие от операции. Операциите в C# могат да бъдат разделени в няколко категории:

- ❖ Двухаргументните *аритметични* операции – събиране, изваждане, умножение и деление – са ни добре познати от математиката и от използването им за съставяне на формули при работа с електронни таблици. По-малко популярна е операцията *намиране на остатък* при делене на цели числа. С използването на аритметичните операции ще се занимаем по-нататък в този Урок.

- ❖ От *операциите за присвояване*, за момента, познаваме само най-простото присвояване с инфиксен знак =. **Много важно е да подчертаем, че присвояването е операция и като всяка операция тя има резултат – стойността, присвоена на променливата в лявата страна.**
 - ❖ Всяка от *операциите за сравнение* проверява дали двата ѝ аргумента са в зададено съотношение или не. Резултатът е една от двете булеви стойности – true, ако проверяваното съотношение е в сила или false, когато съотношението не е в сила.
 - ❖ *Логически операции* се прилагат върху булеви стойности и дават в резултат булеви стойности. Логическите операции конюнкция („и“), дизюнкция („или“) и отрицание („не е вярно, че“) са ни познати от формулите в електронни таблици.
 - ❖ *Побитовите операции* са аналози на логическите, само че се извършват между съответните битове на целочислените типове, като 0 се приема за false, а 1 – за true.
 - ❖ *Други операции*, които е трудно да бъдат класифицирани в някаква група – например, операцията за *слепване* на два низа, знакът на която е + и която също познаваме от работата с електронни таблици.
- Таблица 1 съдържа всички операции на езика C#:

Приоритет	Операции	Асоциативност	
най-висок	<променлива>++, <променлива>--, new, (<тип>), typeof(<променлива>), sizeof(<тип>)	отляво надясно	
	<променлива>++, <променлива>--, +<израз>, -<израз>, !<израз>, ~<израз>	отдясно наляво	
	*, /, %	отляво надясно	
	<низ> + <низ>	отляво надясно	
	+, -	отляво надясно	
	<<, >>	отляво надясно	
	..., <, >, <=, >=, is, as	отляво надясно	
	==, !=	отляво надясно	
	&, ^,	отляво надясно	
	&&	отляво надясно	
...		отляво надясно	
	<израз>?<израз>:<израз>, ??	отдясно наляво	
	най-нисък	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =	отдясно наляво

Фиг. 1. Таблица 1. Операции в C#

В този урок ще разгледаме само аритметичните операции. Ще се запознаем с някои от другите операции в следващи уроци.

Аритметични операции и изрази

Аритметичните изрази са ни познати от математиката и работата с електронни таблици. Построяваме ги от числови стойности, зададени в променливи или като константи, и аритметичните операции. Три от аритметичните операции – събиране (инфиксен знак +), изваждане (инфиксен знак -) и умножение (инфиксен знак *) се извършват еднотипно, независимо какви са аргументите им. Извършването на операцията *деление* (с инфиксен знак /), когато аргументите ѝ са цели числа дава в резултат само цялата част на резултата. Например, резултатът от делението 3/2 е 1, а не 1.5.

За да бъде компенсирана неточността на целочислено деление, в компютрите е реализирана операцията за пресмятане на *остатъка при целочислено деление* (знак %). Например, 3%2 е 1. Остатъкът при деление на 2 често използваме, за да проверим дали едно цяло число е четно или нечетно, например.

На всеки аритметичен израз еднозначно се съпоставя число, което наричаме *стойност* на израза. За да може еднозначно да пресметнем стойността на аритметичен израз, в който имаме повече от една опера-

ция са необходими правила за реда, в който се прилагат операциите. Такъв ред може да зададем с поставянето на кръгли скоби. Например $((3*5)-(7+2))+((4*2)-1)$. Правилото е, че най-напред изпълняваме операция, която е поставена в скоби и в скобите няма други операции. Ако има няколко такива операции в скоби, няма значение от коя ще започнем. Затова изразът от примера пресмятаме в следния ред:

$$((3*5)-(7+2))+((4*2)-1)=(15-9)+(8-1)=6+7=13.$$

Както се вижда от примера, ако редът за прилагане на операциите се определя по този начин, изразите се получават претрупани със скоби. За да се опрости изписването на изразите, се въвеждат допълнителни правила, като *приоритет* и *асоциативност* на операциите:

- ❖ на всяка операция се определя *приоритет*, като между две операции с еднакви приоритети по-рано се изпълнява тази с по-висок приоритет;
- ❖ за множество от операции с еднакъв приоритет се определя *асоциативност* – от ляво на дясно или от дясно на ляво. Когато в израза имаме няколко операции с еднакъв приоритет, те се изпълняват от ляво надясно или от дясно наляво, в зависимост от асоциативността им.

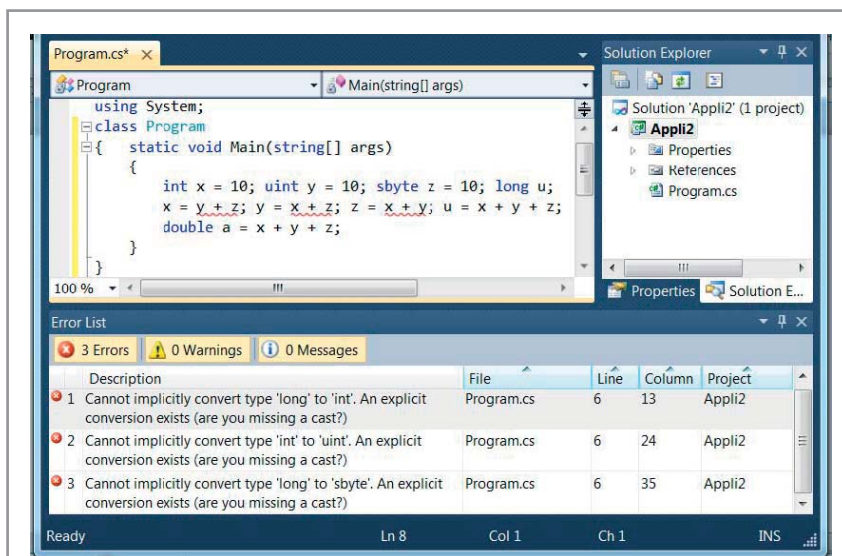
Аритметичните операции умножение, деление и намиране на остатък при целочислено деление, и в математиката и в информатиката, имат по-голям приоритет от събирането и изваждането. Асоциативността и на едните и другите е от ляво на дясно. Затова изразът от примера можем да запишем по-просто така $3*5-(7+2)+4*2-1$. Последните скоби не може да премахнем, защото пресмятайки според асоциативността отляво надясно, ще трябва да извадим 7 и да прибавим 2, което е различно от това да извадим $(7 + 2)$.

В Таблица 1 операциите в един ред са с еднакъв приоритет, като приоритетът по редове намалява отгоре надолу. В най-дясната колона е посочена асоциативността на операциите в съответния ред.

Много важно за еднозначното пресмятане на израз е как компилаторът постъпва, когато в израза участват променливи и/или константи от различни типове. Правилото е, че в операция с два операнда от различен тип, преобразуване на единия тип в другия се допуска само ако стойностите на първия тип са стойности и на втория. Казваме, че вторият тип е *по-мощен* от първия.

Възможно е такова включване на стойности да не е възможно в нито една от двете посоки. Тогава двата типа са *несравними*. Изразът се пресмята само ако в него има стойност от тип, който е по-мощен от всички останали и резултатът е от този тип.

Например, стойност от беззнаковия тип `byte` може да се преобразува както в беззнаковия `uint`, защото интервалът от $[0,255]$ се съдържа в интервала $[0,65535]$, така и в знаковия `int`, защото $[0,255]$ се съдържа и в интервала $[-32768,32767]$. Знаковият `int` и беззнаковият `uint` не могат да се преобразуват един в друг защото, са несравними. Дробните типове `float` и `double` са по-мощни от всеки целочислен тип, т.е. преобразуването ще бъде допуснато от компилатора, но може да се загуби точност. Например, ако присвоите много голяма цяла стойност на променлива от тип `float`. Във всеки случай, когато пресмятането на израза е невъзможно заради несъвместимост на типове, ще получите предупреждение още от редактора на средата (Фиг. 1).



Фиг. 1. Невъзможност за преобразуване на типове

Оператор за присвояване

След като вече знаем какво е израз, можем да дадем по-точна дефиниция на оператора за присвояване.

Синтаксисът на **оператора за присвояване** е:

`<променлива> = <израз> ;`

Действието на оператора е следното: пресмята се изразът отляво на знака за присвояване и стойността му се присвоява на променливата отляво на знака.

Когато изразът се пресмята в оператор за присвояване, това, което ще бъде присвоено на променливата отляво на знака, зависи и от нейния тип. Правилото е следното. Ако пресметнатата стойност на израза е от същия или по-слаб тип от типа на променливата вляво, тогава операторът е допустим. Ако променливата вляво е от по-слаб тип или двата типа са несъвместими, операторът се счита за неправилен.

Работа с компютър:

Разгледайте двете функции от *Фиг. 2*. Опитайте се да определите какво ще бъде изведено в конзолния прозорец при изпълнение на съответните програми.

```
static void Main(string[] args)
{
    double x;
    int y = 10, z = 4;
    x = y / z;
    System.Console.WriteLine(x);
}
```

a.

```
static void Main(string[] args)
{
    double x, y = 10;
    int z = 4;
    x = y / z;
    System.Console.WriteLine(x);
}
```

b.

Фиг. 2.

Създайте конзолно приложение от всяка от двете функции и проверете правилността на предположението си.

Автоинкрементна и автодекрементна операция

Едноаргументната *автоинкрементна* операция е със знак `++`. **Аргумент на операцията може да бъде само целочислена променлива.** Операцията е в два варианта – постфиксен `a++` и префиксен `++a`. Когато операцията се използва самостоятелно в оператор `a++`; или `++a`; тогава двете форми на операцията водят до един и същ резултат – увеличаване на стойността на променлива `a` с 1. В този случай всеки от двата оператора е еквивалентен на оператора за присвояване `a = a + 1`; Когато операцията е включена в израз, например `a++ + b` или `++a + b`, тогава разликата между префиксната и постфиксната форма е съществена. В префиксната форма първо се изпълнява присвояването `a = a + 1` и получената стойност на `a` участва в пресмятането на израза, а при постфиксната – текущата стойност на `a` участва в пресмятането на израза, а след това се изпълнява присвояването `a = a + 1`. Например, ако `a` е 2, а `b` е 3, то след присвояването `c = a++ + b` стойността на `c` ще бъде 5, а стойността на `a` ще бъде 3. Докато след присвояването `c = ++a + b` стойността на `a` също ще бъде 3, но стойността на `c` ще бъде 6.



Автодекрементната операция със знак `--` е напълно аналогична на автоинкрементната, като вместо увеличаване на съдържанието на променливата аргумент с 1 се извършва намаляване на стойността с 1. Например, ако `a` е 2, а `b` е 3, то след присвояването `c = a-- + b` стойността на `c` ще бъде 5, а стойността на `a` ще бъде 1. Докато след присвояването `c = --a + b` стойността на `a` също ще бъде 1, но стойността на `c` ще бъде 4.

Автоинкрементната и автодекрементната операция не са задължителни, но изписването им е по-кратко от алтернативната възможност. Нещо повече, тези две операции се компилират в съответните автоинкрементна и декрементна интструкция на машинния език, които са много по-бързи от общите инструкции за събиране и изваждане.

Въпроси и задачи

1. Какво ще бъде изведено в конзолния прозорец, след изпълнение на програмите?

а. <pre>static void Main(string[] args) { int a = 3; int b; b = a++; System.Console.WriteLine(a); System.Console.WriteLine(b); }</pre>	б. <pre>static void Main(string[] args) { int a = 3; int b; b = ++a; System.Console.WriteLine(a); System.Console.WriteLine(b); }</pre>
в. <pre>static void Main(string[] args) { int a = 3; int b = 5; a--; b++; b=a++ + b; System.Console.WriteLine(a); System.Console.WriteLine(b); }</pre>	г. <pre>static void Main(string[] args) { int a = 3; int b = 5; a--; ++b; a = a + b; System.Console.WriteLine(a); System.Console.WriteLine(b); }</pre>
д. <pre>static void Main(string[] args) { int a = 0, c = 12; int b = 0, d = 5; a = c / d; b = a + b; System.Console.WriteLine(a); System.Console.WriteLine(b); }</pre>	е. <pre>static void Main(string[] args) { int a = 0, c = 12; int b = 10, d = 5; a = c % d; b = b / a; System.Console.WriteLine(a); System.Console.WriteLine(b); }</pre>

2.  Създайте от всяка от програмите в предната задача конзолно приложение, компилирайте го и проверете правилно ли сте определили какво извежда програмата в конзолата.
3.  Напишете конзолно приложение, в което на променливата x се присвоява конкретна стойност, а след това на променливата $suma$ се присвоява сборът от цифрите на трицифрено число, съдържащо се в променливата x .

Класът Math

В предишен урок имахме възможност да споменем ролята на стандартните подпрограми за бързото и качествено програмиране на често решавани задачи. Редица математически функции са трудни за програмиране от неопитни програмисти и затова е по-добре да бъдат включени в програмата като стандартни подпрограми. В езика C# има клас `Math`, който съдържа няколко метода, за пресмятане на математически функции. За да може да ползвате тези методи трябва да включите в програмата си съответното пространство от имена с директивата `using System;`

Атрибути

Класът `Math` има два атрибута:

- ❖ `Math.PI` – именувана константа със стойност числото π – съотношението между дължината на окръжността и нейния диаметър – представено с максималната допустима от типа `double` точност: `const double PI = 3.14159265358979;`
- ❖ `Math.E` – именувана константа със стойност числото e – основа на натуралния логаритъм – представено с максималната допустима от типа `double` точност: `const double E = 2.71828182845905.`

Методи

Класът `Math` предлага много методи за пресмятане на математически функции.

Когато една и съща математическа функция може да се приложи към аргументи от различен тип, тогава в ООП се създава по един метод за всеки от типовете, като всички методи носят едно и също име. За програмиста, нещата изглеждат така сякаш има само един метод, който може да се прилага към аргументи от различен тип. Затова всички едноименни методи ще описваме като един, и ще посочваме множеството от възможните стойности на аргументите – константи или променливи.

Ето някои от най-често използваните методи на класа `Math`:

- ❖ `Math.Abs(<числова стойност/променлива със знак>)` – връща абсолютната стойност на аргумента;
- ❖ `Math.Ceiling(<дробна стойност/променлива>)` – връща най-близката цяла стойност, по-голяма или равна на аргумента. Например `Math.Ceiling(3.14)` е 4;
- ❖ `Math.Floor(<дробна стойност/променлива>)` – връща най-близката цяла стойност, по-малка или равна на аргумента. Например `Math.Floor(3.14)` е 3;
- ❖ `Math.Max(<числова стойност/променлива, числова стойност/променлива>)` – връща по-голямата от двете стойности на аргументите;
- ❖ `Math.Min(<числова стойност/променлива, числова стойност/променлива>)` – връща по-малката от двете стойности на аргументите;
- ❖ `Math.Sqrt(<дробна променлива/стойност от типа double>)` – връща квадратния корен на аргумента;
- ❖ `Math.Pow(<дробна променлива/стойност от типа double>, <дробна променлива/стойност от типа double>)` – връща първия аргумент, повдигнат на степен втория аргумент.

За използването на функциите/методите, които пресмятат стойности е важно да знаем следното правило:

Когато в израз поставим *извикване на функция/метод, който пресмята стойност*, тогава извикването на функция/метода се заменя с пресметнатата стойност и тя участва по-нататък в пресмятане на стойността на израза. Извикването на функция/метод може да се счита като операция с най-висок приоритет.

Примери

Основна част от работата на всяка програма е пресмятането на изрази. Ето няколко примера за пресмятане на изрази, с използване на методите на класа `Math`.

Задача 1. На променливата `s` да се присвои лицето на кръг с радиус `r`.

Решение: `s = Math.PI * r * r;`

Задача 2. На променливата `c` да се присвои дължината на окръжност с радиус `r`.

Решение: `c = 2 * Math.PI * r;`



Задача 3. Стена има правоъгълна форма с височина, съдържаща се в променливата `a` и дължина, съдържаща се в променливата `b`. С една кутия боя може да се покрият с квадратни единици от стената. На променливата `ans` да се присвои броят кутии боя, които трябва да се закупят, за да се боядиса цялата стена.

Решение: `ans = Math.Ceiling((a * b) / c);`

Задача 4. На променливата x да се присвои закръгленото на положително дробно число y . Например, ако y е 3.567, то съдържанието на x трябва да бъде 4, а ако y е 3.499, тогава съдържанието на x трябва да бъде 3.

Решение: $x = \text{Math.Floor}(y + 0.5);$.

Въпроси и задачи

1.  Създайте конзолно приложение, в което декларирайте променливите от примерите в края на урока. Задайте им подходящи стойности, пресметнете изразите от примерите и получените резултати изведете в конзолата.
2.  Метален прът с цилиндрична форма има дължина, съдържаща се в променливата a . От него трябва да се изрежат метални цилиндри с височина, съдържаща се в променливата b . Напишете конзолно приложение, в което се присвояват конкретни стойности на променливите, пресмята се и се извежда броят на цилиндрите, които могат да се изрежат от дадения прът.

15

Въвеждане и извеждане на данни

Работа с компютър

Нека напишем конзолно приложение, което по зададена страна на квадрат намира периметъра и лицето му. Ще ни е необходима една променлива a , в която да бъде съхранена страната на квадрата. Ще се възползваме от възможностите на метода `WriteLine` на класа `Console`, който приема като аргумент произволен израз и ще пресметнем периметъра и лицето в извикването на метода. Затова за периметъра и лицето не са необходими променливи. Да изберем целочислен тип за променливата, в която ще съхраняваме страната на квадрата.

Алгоритъм:

1. Декларираме променлива a за страната на квадрата.
2. Даваме стойност на a .
3. Извеждаме в конзолата периметъра $4*a$.
4. Извеждаме в конзолата лицето $a*a$.

На *Фиг. 1* този прост алгоритъм е представен с оператори на езика `C#`, като четирите реда съответстват на четирите стъпки на алгоритъма.

Създайте конзолно приложение с име `square`. Поставете операторите от *Фиг. 1* в метода `Main`, компилирайте програмата и я изпълнете. Правилно ли намира програмата периметъра и лицето на квадрат със страна 3?

```
int a;  
a = 3;  
Console.WriteLine(4*a);  
Console.WriteLine(a*a);
```

Фиг. 1.

Мемогъм Write

Програмата `square` очевидно има сериозен недостатък. При всяко стартиране тя извежда в конзолата две числа, всеки път едни и същи, за които потребителят даже не знае какво представляват. Редно е изходът на програмата да е по-разбираем и да подсказва за какво се отнасят извежданите данни. Затова може да я подобрите, като добавите извеждане на подсказващи текстове така, както е показано на *Фиг. 2*.

Всъщност извеждането на подсказващите текстове на отделен ред от съответните стойности е ненужно

```
int a;  
a = 3;  
Console.WriteLine("Страна:");  
Console.WriteLine(a);  
Console.WriteLine("Периметър:");  
Console.WriteLine(4*a);  
Console.WriteLine("Лице:");  
Console.WriteLine(a*a);
```

Фиг. 2.

```
int a;
a = 3;
Console.Write("Страна: ");
Console.WriteLine(a);
Console.Write("Периметър: ");
Console.WriteLine(4*a);
Console.Write("Лице: ");
Console.WriteLine(a*a);
```

Фиг. 3.

удължаване на изхода, но както показва и името на метода `WriteLine` (напиши/изведи един ред), такова е действието му – след като изведе зададения му аргумент, преминава на нов ред. Класът `Console` има метод `Write`, който се отличава от метода `WriteLine` по това, че след извеждане на аргумента не премества показалеца на нов ред, а го остава след последния изведен знак. Заменете в програмата `square` метода `Write` с `WriteLine`, както е показано на Фиг. 3. Компилирайте програмата и я изпълнете.

С направеното дотук изходът на програмата доби по-добър вид, но тя продължава да пресмята само пери-

метър и лицето на квадрат със страна 3. Разбира се, може да заменим оператора `a = 3;` с оператора `a = 12;`, да компилираме отново и да пресметнем периметъра и лицето на квадрат със страна 12, но това не е добро решение. Както споменахме в урока за Алгоритми, редно е един алгоритъм да може да бъде изпълнен за колкото може повече различни входни данни, без да компилираме отново и отново програмата. За тази цел обаче трябва да може да задаваме на програмата данните, с които искаме да се изпълни.

Memogume ReadLine u Parse

Класът `Console` има метод `ReadLine()` за въвеждане на данни от клавиатурата. Когато изпълнението на програмата достигне до извикването на този метод, програмата спира и изчаква потребителя да напише това, което иска и да натисне клавиша `Enter`. От знаците, въведени от потребителя, методът образува **низ**. За да стане достъпен този низ в програмата, програмистът трябва да декларира променлива от тип `string` и да присвои на тази променлива резултата от извикването на метода.

Методът `ReadLine()` има една особеност – той въвежда написаното от потребителя на клавиатурата в променлива от тип `string`. Така например, ако потребителят напише числото 12, методът ще въведе в променливата от тип `string`, зададена отляво на знака за присвояване, низа "12". За да преобразуваме този низ в целочислената константа 12, използваме метода `Parse(<низ>)` на класа `int`, който преобразува зададения му като аргумент низ в цяло число. Забележете, че `Parse()` не е метод на класа `Console`, а на клас, който не сме срещали досега – класа `int`. Например, `int a; a = int.Parse(s)`.

Работа с компютър

```
string s;
Console.WriteLine("Въведете име:");
s = Console.ReadLine();
Console.Write("Здравей ");
Console.Write(s);
Console.WriteLine("!");
```

Фиг. 4.

Ще илюстрираме използването на `ReadLine`, като създадем конзолно приложение с име `greeting`, което прочита от клавиатурата името, зададено му от потребителя и отправя поздрав. Поставете операторите от Фиг. 4 в метода `Main`, компилирайте и изпълнете програмата няколко пъти, като след извеждането на подсказката "Въведете име:" въведете различни имена.

Тук е мястото да въведем операцията *сливане* (конкатенация) на два низа. Знак на операцията е знакът

плюс. Сливането $s_1 + s_2$ на низовете s_1 и s_2 дава нов низ s , в който s_1 и s_2 са изписани един след друг. Например, ако s_1 е "Hello ", а s_2 е "world!", то $s = s_1 + s_2$ е низът "Hello world!". Низът s_1 е *начало* (или *префикс*) на s , а s_2 е край (или *суфикс*) на s .

Заменете последните три реда в програмата `greeting` с реда

```
Console.WriteLine("Здравей " + s + "!");
```

Компилирайте и изпълнете програмата.

Работа с компютър

Програмата от Фиг. 5 реализира идеята, страната на квадрата да се въвежда от потребителя. За да е ясно на потребителя, който ще я използва какво е предназначението ѝ, сме добавили кратка ин-

струкция с използването на метода `WriteLine`. Необичайно тук е само извикването `WriteLine("")`. Не е трудно да се досетите, че извикан с празния стринг `""` методът ще предизвика само преминаване на нов ред, тъй като няма какво да извежда в реда.

Променете конзолното приложение `square`, като не изтривате старото съдържание на функцията `Main`, а я редактирате така, че да получите показаното на *Фиг. 5*. Използвайте командите `Copy` и `Paste`, за да направите копия на близки по съдържание оператори и след това ги редактирайте. Компилирайте програмата и я изпълнете.

При стартиране на програмата, в конзолата ще се появи показаното на *Фиг. 6* и в края на последния изведен ред – мигащ показалец (няма го на фигурата). Програмата чака от вас да въведете данни. Напишете, например `3415` и натиснете `Enter`. Проверете с калкулатора правилен ли е изведеният от програмата резултат.

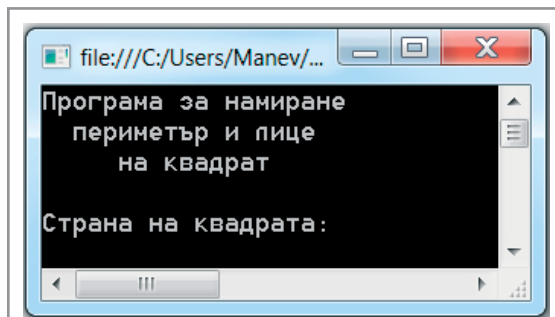
Изпълнете програмата отново, като вместо число въведете произволни знаци, например букви. Забележете как реагира програмата – *прекъсна* изпълнението *аварийно*, поради случило се *извънредно събитие*, при което не е предвидено какво да направи програмата (`unhandled exception`) и изведе съобщение, че въвежданите данни не са в очаквания формат.

Едно от задълженията на програмиста е да предвиди действията на потребителя и да се опита да напише така програмата, че да не прекъсва аварийно работата си, да не извежда многобройни, неразбираеми за потребителя съобщения или да работи, но да дава грешни резултати. За целта програмата трябва да **обработва грешките**, възникнали по време на работа.

Въвеждането на знаци, различни от цифри не е единствения проблем на програмата. Опитайте да намерите и други некоректни въвеждания. До следващ урок, когато ще поправим недостатъка на програмата – липса на реакция при некоректното въвеждане на входните данни.

```
int a; string s;
Console.WriteLine("Програма за намиране");
Console.WriteLine(" периметър и лице ");
Console.WriteLine(" на квадрат ");
Console.WriteLine("");
Console.Write("Страна на квадрата:");
s=Console.ReadLine();
a=int.Parse(s);
Console.Write("Обиколката е ");
Console.WriteLine(4*a);
Console.Write("Лицето е ");
Console.WriteLine(a*a);
```

Фиг. 5.



Фиг. 6.

Въпроси и задачи

1. Каква е разликата между методите `Write` и `WriteLine`?
2. Вече познаваме метода `Parse` на класа `int`. Разгледайте фрагмента от програма:

```
a = double.Parse(c);
b = long.Parse(d);
```

Предположете от какъв тип са променливите `a`, `b`, `c` и `d`.
3. Както се вижда в предишната задача, метод `Parse` имат и класовете `double` и `long`. Първият ще ви позволи да намирате периметър и лице на квадрати, страните на които са дробни числа, а вторият – на квадрати страните на които са много по-големи от допустимите за типа `int`. Напишете версии на програмата `square`, в които страната на квадрата е дробно или много голямо цяло. Компилирайте и тествайте тези програми.
4. Съхранете всички файлове на приложението `square` под името `circle`. Променете кода на програмата така, че да въвежда радиуса на кръг и да извежда периметъра и лицето му.
5. Съхранете всички файлове на приложението `square` под името `rectangle`. Променете кода на програмата така, че да въвежда двете страни страни на правоъгълник и да извежда периметъра и лицето му.

Както вече знаем, общуването на потребителя и работещата програма може да стане по различен начин – с графичен интерфейс, с буквено-цифров интерфейс (както при конзолните приложения, в които в предишния урок се научихме да въвеждаме данни) или по други начини. Разбира се, че програмите с графичен интерфейс са по-удобни за обикновения потребител, по-привлекателни, и постепенно изместиха програмите с буквено-цифров интерфейс. Въпреки това, конзолата си остава незаменимо средство за общуване с програмата в някои случаи.

Един от тези случаи е необходимостта от малки програми, със сравнително прост вход, за еднократна употреба, когато е необходимо вниманието да се насочи към конкретния проблем, който решаваме, а не към елегантно представяне на резултатите. Друг случай, в който конзолата е полезна е, когато тестваме част от кода на голяма програма. Поради простотата на работа на конзолното приложение, може да се изолира тази част от кода лесно и удобно, без да се налага да се преминава през сложен потребителски интерфейс и поредица от прозорци, за да се стигне до желания код за тестване.

Форматиран изход

За да се направи конзолното приложение по-удобно за потребителя, е необходимо извежданите данни да са форматираны по подходящ начин. За разлика от въвеждането, където всички данни попадат в програмата във вид на низ и трябва да се преобразуват с метода `Parse`, при извеждането могат да се извеждат директно константи, променливи и стойности на изрази от всички познати типове чрез познатите ни методи `Console.WriteLine(<израз>)` и `Console.Write(<израз>)`. Да напомним разликата между двата метода: методът `Write` извежда на конзолата стойността на израза, докато методът `WriteLine` прави същото, след което преминава на нов ред.

Да разгледаме ситуация, при която се налага да се изведат на един ред на конзолата данни от различни типове. Например, в променливата `years` е пресметната възрастта на потребителя и програмата трябва да изведе съобщението: "Вие сте на ... години", където на мястото на точките трябва да се постави съдържанието на променливата `years`. Вече знаем два начина да направим това:

Първият начин е да разделим текста на три части, които да изведем с три отделни извиквания на методи:

```
Console.Write("Вие сте на ");
Console.Write(years);
Console.WriteLine(" години");
```

Вторият начин е, да използваме операцията сливане на низове:

```
Console.WriteLine("Вие сте на " + x + " години");
```

Сега ще покажем и трета възможност да направим същото, като покажем специалните възможности на методите `WriteLine` и `Write` за форматиране на изхода.

Всъщност съществуват методи `WriteLine` и `Write` в следния по-общ вид

```
WriteLine (<форматиращ низ>, <списък от изрази>)
Write (<форматиращ низ>, <списък от изрази>)
```

където

- ❖ <списък от изрази> се състои от няколко израза, разделени един от друг със запетаи, номерирани с 0, 1, 2 и т.н, в реда по който се срещат в списъка;
- ❖ <форматиращ низ> е константен низ, съдържащ текста, който ще се извежда и *форматиращи елементи*. Най-простият вид на форматиращ елемент е {<номер на израз>}. Това означава, че на мястото на форматиращия елемент ще бъде изведена стойността на израза, с посочения във форматираща елемент номер в списъка от израза.

Например, ако в променлива `lev` сме запомнили левовата част на някаква сума, а стотинките – в променливата `st` и искаме програмата да изведе съобщение за тази сума, тогава можем да напишем оператора:

```
Console.WriteLine("Сумата е {0} лева и {1} стотинки", lev,st);
```

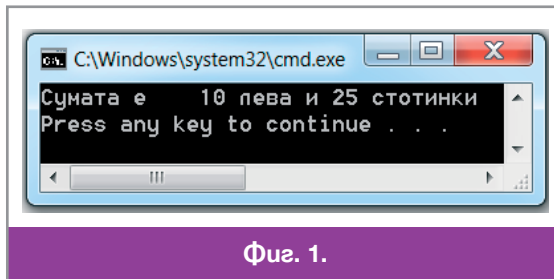
Така, на мястото на форматиращия елемент `{0}` ще бъде изведена стойността на променливата `lev`, на мястото на форматиращия елемент `{1}` ще бъде изведена стойността на променливата `st`.

Ако във форматиращия елемент, след номера на израз и разделено със запетая от него, е изписано още едно число, то задава броя позиции, в които да се изведе стойността на посочения израз. Ако извежданата стойност заема по-малко позиции, тогава в излишните позиции ще се изведат интервали. Ако броят позиции е недостатъчен, ще бъдат заделени автоматично толкова позиции, колкото трябва.

Например, резултатът от изпълнение на програмата:

```
static void Main(string[] args)
{
    int lev=10, st=25;
    Console.WriteLine("Сумата е {0,5} лева и {1,1} стотинки", lev,st);
}
```

е показан на *Фиг. 1*. Тъй като променливата `lev` има стойност, записваща се в 2 знака, вляво остават 3 интервала. Стойността на променливата `st` пък не се събира в един знак и затова програмата отделя за нея необходимите 2 знака.



Фиг. 1.

Управляващи знаци

Управляващите знаци са с кодове от 0 до 31 в таблицата Unicode. Някои от тях се използват за форматиране на изхода при методите за извеждане. Тъй като тези знаци нямат свое графично изображение, прието е да се изписват с някакъв знак, напоянящ за предназначението им при форматиране на извода, предшестван от знака обратна наклонена черта (`\`). В следната таблица са дадени някои управляващи знаци:

<code>'\n'</code> – премини на нов ред (new line)	<code>'\''</code> – изведи апостроф
<code>'\a'</code> – издай звук (system beep)	<code>'\"'</code> – изведи кавички
<code>'\b'</code> – изтрий последния знак (backspace)	<code>'\{'</code> – изведи лява фигурна скоба
<code>'\r'</code> – върни в началото на реда (return)	<code>'\}'</code> – изведи дясна фигурна скоба
<code>'\t'</code> – хоризонтална табулация (tab)	<code>'\\'</code> – изведи обратна наклонена черта

Така например, извикването `Console.Write("{0}\n",x)` е еквивалентно на извикването `Console.WriteLine(x)`.

Забележете, че кавичките, апострофите и фигурните скоби играят специална роля при оформяне на форматиращия низ. Затова когато се налага да изведем някой от тези знаци на конзолата – трябва да отменим специалното му значение. Това става, като пред него напишем знака обратна наклонена черта (`\`). В тази си роля знакът обратна наклонена черта става специален и затова когато трябва да го изведем на конзолата – изписваме две обратни наклонени черти.

Например, ако искаме да изведем на конзолата текста "Hello world" е любим поздрав на програмистите, не можем да я изпишем в този вид между двойка кавички. Опитайте да го направите и вижте какво ще се случи. Низът трябва да се изпише така, както е показано на *Фиг. 2a* и тогава резултатът ще бъде този който искаме (*Фиг. 2б*).

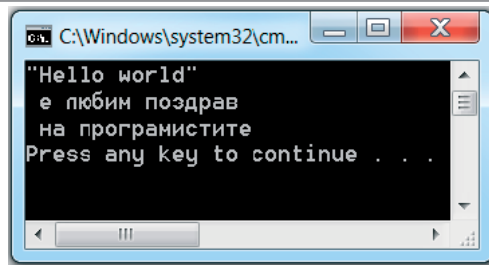
Ако пред форматиращия низ се постави знакът `"@"`, то зададеният текст ще се изведе така, както е зададен в редактора, дори и да е на повече от един ред, заедно с включените интервали, знаци за табулация и т.н. Например, това което ще изведе програмата от *Фиг. 3a*, е показано на *Фиг. 3б*.


```

class Program
{
    static void Main(string[] a)
    {
        Console.WriteLine("\nHello world"
            + "\n е любим поздрав\n"
            + " на програмистите\n");
    }
}

```

а.



б.

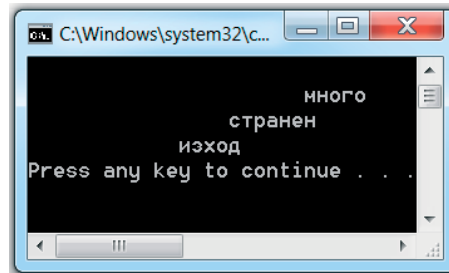
Фиг. 2.

```

static void Main(string[] args)
{
    Console.WriteLine(@"
        много
        странен
        изход");
}

```

а.



б.

Фиг. 3.

Работа с компютър

Да напишем програма, която разменя стойностите на две променливи. Това може да бъде извършено по няколко начина. Два оператора: $a = b$; $b = a$; на пръв поглед правят това, но ако се вгледаме по-внимателно и си припомним същността на операцията присвояване, ще разберем, че тези два оператора не водят до искания резултат. Напишете конзолно приложение, което въвежда стойности в a и b , изпълнява оператори: $a = b$; $b = a$; и показва получения резултат. Причината за неуспеха е, че когато присвоим на a стойността на b , изгубваме стойността на a .

От тези разсъждения се вижда, че за да не загубим стойността на a , ще ни трябва още една променлива c , която преди присвояването $a = b$; да съхрани стойността на a , и от която след това ще прехвърлим първоначалната стойност на a в b . Въведете получената програма (Фиг. 4), съхранете я под името `swap`, компилирайте я и я изпълнете, за да проверите работоспособността ѝ.

Ако правилно сте въвели програмата, резултатът от изпълнението трябва да бъде този, който е показан на Фиг. 5. Има и друг интересен начин за размяна стойностите на две променливи, при който

```

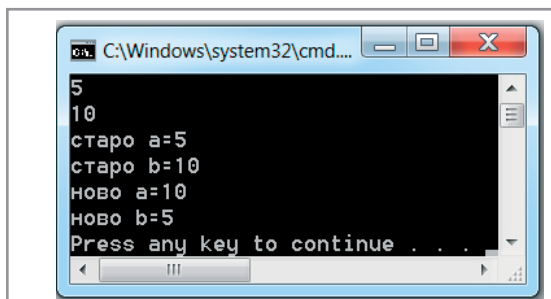
using System;
class Program
{
    static void Main(string[] args)
    {
        int a, b, c; string s;
        s=Console.ReadLine(); a = int.Parse(s);
        s=Console.ReadLine(); b = int.Parse(s);
        Console.WriteLine("старо a={0}/n старо b={1}", a, b);
        c = a; a = b; b = c;
        Console.WriteLine("ново a={0}/n ново b={1}", a, b);
    }
}

```

Фиг. 4.

не се използва допълнителна променлива, а операторите за присвояване: $a = a + b$; $b = a - b$; $a = a - b$;

Напишете нова версия на програмата с име `swap1`, която реализира този подход, компилирайте я и проверете работоспособността ѝ. Този подход, обаче, не винаги е за предпочитане, тъй като е малко по-бавен, а при големи стойности на променливите `a` и `b` може да доведе до *препълване* на типа, т.е. получаване на по-голяма стойност от максималната, която променлива от използвания тип може да съдържа (в кой от трите оператора може да стане това?)!



Фиг. 5.

Въпроси и задачи

1. Напишете програма, която извежда на екрана името на вашето училище, оградено с кавички.
2. Довършете оператора

```
Console.WriteLine("Всеки . . . идентифицира.");
```

като замените многоточието с текст така, че в конзолата да се изведе:

```
Всеки гражданин има  
ЕГН\ЛНЧ, чрез който се идентифицира.
```

3. Напишете конзолно приложение, което да изведе на екрана следния текст:

```
След изпълнението на операторите:  
int a = 5;  
if (a > 0) { a = a + 5; }  
else { a = a - 5; }  
стойността на променливата 'a' ще бъде 10.
```

4. Допишете програмния фрагмент

```
int a = 10, b = 12, c = 14;  
Console.WriteLine(. . . . .);
```

така, че, стойностите на трите променливи да бъдат изведени в един ред на конзолата, разделени със знака за хоризонтално табулиране, т.е.

```
10      12      14
```

17

Примери за конзолни приложения

Задача 1. Да се състави програма `muldig`, която въвежда от клавиатурата едно положително двуцифрено число и извежда произведението на цифрите му.

ПРИМЕР:	Вход:	Изход:
	23	6

Решение: Единствената трудност на задачата е отделянето на двете цифри на въведеното число. От урока за представяне на числата в позиционни бройни системи знаем, че цифрата на единиците на двуцифрено число, представено в десетична бройна система, е остатъкът от целочисленото деление на числото на 10, а цифрата на десетиците – частното от това деление.

Програма:

```
using System;  
namespace ConsoleApplication1  
{  
    class Program  
    {  
        static void Main(string[] args)
```

```

    { Console.WriteLine("Въведи двуцифрено число:");
      string s;
      s = Console.ReadLine();
      int number;
      number = int.Parse(s);
      int pr;
      pr = (number % 10) * (number / 10);
      Console.WriteLine("Произведението е {0}",pr);
    }
  }
}

```

Въведете програмата, компилирайте я и проверете работоспособността ѝ с няколко примера. Какъв е резултатът от изпълнението на програмата ако зададете едноцифрено число? А ако зададете трицифрено?

Задача 2. Напишете програма `exchng`, която въвежда трицифрено цяло положително число N и извежда числото, което се получава, когато разменим местата на първата и последната цифра на N .

ПРИМЕР:	Вход:	Изход:
	123	321

Решение: Първата и последната цифра на трицифрено число можем да получим както и в **Задача 1** като частното от целочисленото деление на числото на 100 и остатъкът от такова деление на числото на 10, съответно. Проблемът в този алгоритъм е отделянето на втората цифра, която ще ни трябва за построяване на новото число. За целта отделяме двуцифреното число, съставено от първите две цифри като частното от деленето на 100, а от него отделяме цифрата на единиците му, като остатък при деление на 10.

Програма:

```

using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Въведете трицифрено число:");
        string s = Console.ReadLine();
        int number = int.Parse(s);
        int ed, des, sto;
        ed = number % 10; //цифрата на единиците
        des = (number / 10) % 10; //цифрата на десетиците
        sto = number / 100; //цифрата на стотиците
        int swap; //размяна на цифрите
        swap = ed; ed = sto; sto = swap;
        int newNumber;
        newNumber = sto * 100 + des * 10 + ed; //новото число
        Console.WriteLine("Новото число е {0}", newNumber);
    }
}

```

Въведете програмата, компилирайте я и проверете работоспособността ѝ с няколко примера. Какъв е резултатът от изпълнението на програмата, ако зададете двуцифрено число? А ако зададете едноцифрено?

Задача 3. Напишете програма `excellent`, която въвежда броя на учениците в един клас и броя на тези ученици, които са получили отлична оценка на контролна работа по информатика, след което намира процента на отличниците, закръглен до втората цифра след десетичната точка.

Пример:	Вход:	Изход:
	26	38.46
	10	

Решение: Трудността в тази задача е изрязването на дробно число до втория знак след десетичната точка. За целта ще ползваме форматиращ елемент от вида `{<номер на израз>: .##..#}`, който при извеждане на число взема от дробната му част толкова цифри, колкото са знаците `#` във форматиращия елемент.

Програма:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Въведете броя на учениците:");
        string s = Console.ReadLine();
        int numberStudents = int.Parse(s);
        Console.WriteLine("Въведете броя на отличниците:");
        s = Console.ReadLine();
        double numberExcellent = double.Parse(s);
        double percent = (numberExcellent/numberStudents) * 100;
        Console.WriteLine("Процент отличници {0:.##}", percent);
    }
}
```

Задача 4. Напишете програма, която въвежда часа и минутите от началото на часа на излитането на самолет, както и продължителността на полета му в минути. Програмата трябва да изведе часа и минутите от началото на часа, когато самолетът трябва да кацне, при условие, че кацането става в деня на излитане.

ПРИМЕР:	Вход:	Изход:
	10	12h 30min
	51	
	99	

Решение: Използваме обичайната техника за работа с мерни единици, когато всички данни се превръщат в най-малката от използваните мерни единици – в случая минути, извършват се пресмятанията, а резултатът отново се преобразува в началния вид, в случая – час и минути от началото на часа.

Програма:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Въведи час на излитане:");
        string s = Console.ReadLine();
        int hours = int.Parse(s);
        Console.WriteLine("Въведи минути на излитане:");
        s = Console.ReadLine();
        int minutes = int.Parse(s);
        Console.WriteLine("Продължителност на полета в минути:");
        s = Console.ReadLine();
        int x = int.Parse(s);
        int allMinutes = hours * 60 + minutes + x;
        int newHours = allMinutes / 60;
        int newMinutes = allMinutes % 60;
        Console.WriteLine("{0}h {1}min", newHours, newMinutes);
    }
}
```

Условен оператор

В предишен урок написахме програма `square`, с която въвеждахме страната на квадрат и извеждахме периметъра и лицето му. Когато въведохме в програмата буква вместо цифра, тя завърши работа аварийно. Там поставихме задачата за намиране на входни данни, които ще доведат до грешен резултат.

тат? Например: „Дължина на страната на квадрата, която е по-малка или равна на нула“. Не може да има квадрат с дължина на страната 0 или отрицателно число! Ако стартираме програмата и въведем дължина на страната -5, ще получим за лицето на несъществуващ квадрат $25 = (-5) \cdot (-5)$ и невъзможния периметър $-20 = (-5) \cdot 4$. Това е недопустимо! Програмата трябва да е направена така, че да разпознава погрешно въведените данни и да предупреждава за това потребителя, а не да завършва аварийно или да извежда безсмислен резултат.

Какво трябва да се промени в програмата? Проблемът е в това, че след въвеждането на дължината на страната, не се проверява дали въведеното число е положително. Редно е, след оператора, в който въведеният низ s се преобразува в числото a , програмата да „се запита“ дали a е по-голямо от 0. Ако това е вярно – да продължи с пресмятането на периметъра и лицето. Иначе, да изведе съобщение за грешка и да спре. Казано накратко:

ако $a > 0$ пресмятай, иначе съобщи за грешен вход.

В общия случай, за такава ситуация ще трябва да включим в алгоритъма на програмата указание от вида

ако е изпълнено някакво условие ► направи нещо,

иначе ► направи друго нещо.

Такава конструкция в езика C# реализираме с *условния оператор*.

Синтаксисът на *условния оператор* е:

```
if (<условие>
{ <оператори за случая, когато условието е изпълнено> }
else
{ <оператори за случая, когато условието не е изпълнено> }
```

Думите `if` (англ. ако) и `else` (англ. иначе, в противен случай) са служебни думи и, както знаем, не могат да се използват като имена. Фигурните скоби се пишат задължително само когато, в съответния случай, трябва да се изпълни повече от един оператор. Ако операторът е само един – скобите може да се пропуснат, но не съветваме начинаещите програмисти да го правят – със скоби текстът на програмата се чете по-лесно. Няколко оператора, поставени във фигурни скоби, наричаме *блок от оператори*.

Условният оператор има вариант, при който може да се изпусне блокът от оператори след `else`, т.е. ако условието е в сила, се изпълняват операторите от блока след `if`, в противен случай изпълнението на програмата продължава с оператора, който е непосредствено след *условния*.

Условия

Знак	Значение
==	Равно (=)
>	По-голямо (>)
<	По-малко (<)
>=	По-голямо или равно (\geq)
<=	По-малко или равно (\leq)
!=	Не равно (\neq)

Фиг. 1. Сравнения

Когато програмата трябва да вземе някакво решение, това става с помощта на *условие*. Условието е булев израз, стойността му е `true` (вярно) или `false` (невярно). В *условния оператор*, както и на всяко друго място в програмата, където условието е задължително, **булеви-ят израз се поставя в кръгли скоби**.

Най-простият начин за построяване на условие е с помощта на операциите за сравняване, които познаваме от уроците за електронни таблици. На *Фиг. 1* са показани знаците на шестте операции за сравняване в езика C#, като знаците на две от операциите се различават от използваните в MS Excel. Кои са те?

Двата аргумента на операция за сравняване може да са произволни изрази, стойностите на които могат да се сравняват. Например, може да се сравняват стойности от кои да са два числови типа, но не може да се сравняват числови стойности с низове. Стойностите от тип `char` се сравняват както числа, защото в действителност се сравняват техните кодове в таблицата Unicode.

Изразите от тип `string` в C# могат да участват само в сравнения от типа `==` и `!=`. Както знаем от уроците за електронни таблици, другите сравнявания на низове също имат смисъл, когато се използва

лексикографската наредба. Причината да не са позволени в C# е, че в таблицата Unicode има всевъзможни азбуки и резултатът от сравняването може да е безсмислен.

За пример, нека разгледаме трите програмни фрагмента от *Фиг. 2*:

<pre>int a, b, c; a=12; b=6; if (a>b) a=b; c=a+b; Console.WriteLine(c);</pre>	<pre>int a, b, c; a=12; b=6; if (a<b) a=b; c=a+b; Console.WriteLine(c);</pre>	<pre>int a, b, c; a=12; b=6; if (a>b) a=b; else a=20; c=a+b; Console.WriteLine(c);</pre>
Фрагмент 1	Фрагмент 2	Фрагмент 3

Фиг. 2.

Във Фрагмент 1 операторът `if (a>b) a=b;` проверява дали $12 > 6$, което е вярно. Затова на `a` се присвоява стойността на `b` и двете променливи имат стойност 6. С оператора `c=a+b` на променливата `c` се присвоява сумата на `a` и `b`, т.е. $6 + 6 = 12$ и това е стойността, която програмата извежда в конзолата.

Във Фрагмент 2 операторът `if (a<b) a=b;` проверява дали $12 < 6$, което не е вярно. Затова не се изпълнява присвояването `a=b;` и стойностите на `a` и `b` остават същите. Операторът `c=a+b` присвоява на `c` сумата на `a` и `b`, т.е. $12 + 6 = 18$ и това е стойността, която програмата извежда.

Във Фрагмент 3 отново се проверява $12 < 6$, което не е вярно. Затова се изпълнява операторът след `else` и стойността на променливата `a` става 20. Операторът `c=a+b` присвоява на `c` сумата на `a` и `b`, т.е. $20 + 6 = 26$ и програмата извежда тази стойност.

Булев тип

Един от типове, с които се запознахме в предишен урок, беше *булевият тип*. Променливи от булев тип декларираме с ключовата дума `bool`. На променливи от този тип можем да присвояваме стойностите на булеви изрази, които искаме да използваме като условия. За пример да разгледаме фрагмента:

```
int a; bool b;
a = 10;
b = (a < 100);
```

Тъй като съдържанието на променливата `a` е по-малко от 100, след изпълнението на фрагмента, променливата `b` ще има стойност `true`. При присвояване на стойност на логическа променлива, булевият израз не е нужно да се поставя в скоби. След като на променливата `b` е присвоена булева стойност, тя може да участва навсякъде, където се изисква условие. Например: `if (b) a = 200;`.

Работа с компютър

Да се върнем към програмата `square` от предишен урок, с която пресмятахме лице и периметър на кръг. Сега вече може да разрешим проблема с въвеждането на отрицателните числа. Разгледайте кода на *Фиг. 3*. и го сравнете с кода на старата програма.

Забележете, че когато въведената дължина на страната е положителна, трябва да се изпълнят няколко оператора, затова те **задължително** се поставят меж-

```
a = int.Parse(s);
if (a > 0) //ако условието е изпълнено
{
    Console.Write("Периметърът е ");
    Console.WriteLine(4 * a);
    Console.WriteLine("");
    Console.Write("Лицето е ");
    Console.WriteLine(a * a);
    Console.WriteLine("");
}
else //ако условието не е изпълнено
    Console.WriteLine("Грешно въведена страна !");
```

Фиг. 3.

ду две фигурни скоби. В частта `else` има само един оператор, затова тя може се постави между две скоби, но може и да се изпише без скоби, както е във фрагмента.

Създайте конзолно приложение с показания на *Фиг. 3*. Компилирайте програмата и я тествайте. Пробвайте както положителна дължина, така и нулева и отрицателна. Проблемът с въвеждане на букви вместо цифри остава. И той е разрешим, но не със знанията, получени досега. Затова ще оставим програмата в този вариант и ще се върнем към нея в следващ урок.

Въпроси и задачи

1. Каква ще е стойността на променливата `a` след изпълнение на всеки от фрагментите:

а.

```
a = -10;
if (a > 0)
{a = a+10;
}
else
a = a-2;
```

б.

```
a = -10;
b = 5;
if (a > 0)
{ a = a+b;
  a = a*2;
}
a = a-2;
```

в.

```
a = -10;
b = 5;
if (a > 0)
    a = a+b;
a = a*2;
a = a-2;
```

19

Програми с условен оператор

Вложени условни оператори

Понякога се налага, в един от двата случая – в `if` частта или в `else` частта да се напише нов условен оператор. В такъв случай казваме, че имаме *вложен условен оператор*. Да разгледаме няколко примера, в които е използван вложен условен оператор (наричан за по-кратко „вложен `if`“).

Пример 1:

```
a = -3; b = 3;
c = 0;
if (a > 0)
    if (b > 0)
        c = 20;           //този ред ще се изпълни когато a>0 и b>0
    else
        c = -20;         //този ред ще се изпълни когато a>0 и b<=0
Console.WriteLine(c);   //резултат: c=?
```

Опитайте се да определите какво ще изведе в конзолата програмата, съдържаща този програмен фрагмент.

Следващите два примера се различават от **Пример 1**, само с наличието на скоби. За всеки от тях предположете какво ще изведе програмата в конзолния прозорец.

Пример 2:

```
a = -3; b = 3;
c = 0;
if (a > 0)
{ if (b > 0)
    c = 20;           //този ред ще се изпълни когато a>0 и b>0
```

```

else
    c = -20;        //този ред ще се изпълни когато a>0 и b<=0
}
Console.WriteLine(c);    //резултат: c=?

```

Пример 3:

```

a = -3; b = 3;
c = 0;
if (a > 0)
{
    if (b > 0)
        c = 20;        //този ред ще се изпълни когато a>0 и b>0
}
else
    c = -20;        //този ред ще се изпълни когато a>0 и b<=0
Console.WriteLine(c);    //резултат: c=?

```

Работа с компютър

1. Въведете всеки от трите фрагмента и направете от него конзолно приложение. Компилирайте го и проверете дали предположението за това, какво извежда фрагментът в конзолата, е правилно. Опитайте се да обясните разликата в получените резултати от изпълнението на фрагмента в **Пример 2** и фрагмента в **Пример 3**.

2. В предишен урок поставихме задачата да се промени програмата square в програма rectangle така, че да намира периметъра и лицето на правоъгълник. Сега да подобрим програмата rectangle така, че да не допуска пресмятането на периметъра и лицето, ако въведената дължина на една от двете страни не е положителна.

Проверката за коректност на входните данни в този случай е по-сложна, защото трябва да се проверят стойностите на две променливи. Първо да забележим, че няма смисъл потребителят да въвежда дължината на втората страна, ако е въвел грешно дължината на първата. Или по-точно: ако потребителят въведе некоректна дължина на първата страна, програмата трябва да изведе съобщение за грешка и да преустанови работа. Когато, обаче, първата дължина е коректна, от потребителя се иска да въведе втората. За нея също ще има проверка дали е положителна – ако да, т.е. и двете дължини на страни са коректни, програмата може да пресметне и изведе резултата. В противен случай програмата ще трябва да съобщи, че втората дължина е грешна и да спре работа.

Получаваме алгоритъма, показан на *Фиг. 4*.

въвеждане на дължината на страната <i>a</i>	
ако $a > 0$	
въвеждане на дължината на страната <i>b</i>	
ако $b > 0$	изпълнява се когато $a > 0$ И $b > 0$
извеждане на периметъра и лицето	
иначе	изпълнява се когато $a > 0$ И $b \leq 0$
извеждане на съобщение за некоректно <i>b</i>	
иначе	изпълнява се когато $a \leq 0$
извеждане на съобщение за некоректно <i>a</i>	

Фиг. 4.

Напишете конзолно приложение, което реализира алгоритъма. Компилирайте го и проверете работоспособността му. Може да направите малка промяна в алгоритъма – след въвеждане на дължината на страната *a* веднага да се въведе и дължината на страната *b*. Но в такъв случай, при недопустима стойност на *a*, въвеждането на *b* ще се окаже излишно, защото програмата няма да изчисли периметъра и лицето на правоъгълника.

От уроците за електронни таблици познаваме логическите операции И (AND), ИЛИ (OR) и НЕ (NOT). Забележете, че в обяснението на алгоритъма използвахме операцията И. В езика C# операциите И, ИЛИ и НЕ се означават с &&, || и !, съответно. Ще се запознаем по-подробно с тях в следващи уроци, но по-смелите могат вече да опитат да напишат условия, включващи тези операции.

Въпроси и задачи

1. Каква ще е стойността на променливата *a* след изпълнение на фрагмента от *Фиг. 5*?
2. Напишете конзолно приложение, което въвежда показанията на електронен часовник – текущ час и минути от началото на текущия час – и проверява дали са въведени правилно. Часът трябва да е в интервала от 0 до 23, а минутите – от 0 до 59.
3. Подът на стая ще се покрива с паркет. Дадени са размерите на стаята и колко е цената на 1 кв.м. паркет. Напишете конзолно приложение, което въвежда данните и изчислява стойността на необходимия паркет. Програмата трябва да прави проверка на входните данни и при некоректни входни данни да извежда в конзолата подходящо съобщение.
4. Напишете конзолно приложение, което въвежда цяло число и извежда **Да**, ако числото е четно, или **Не** в противен случай.
5. Напишете конзолно приложение, което въвежда цяло число и извежда в конзолата подходящо съобщение за това, дали числото е по-малко от, равно на или по-голямо от нула.

```
a = 10;
if (a > 0)
{
    Console.Write(a+2);
}
else
    Console.WriteLine(a-2);
```

Фиг. 5.

20

Тест: Променливи. Условен оператор

1. Кои от следните низове НЕ са правилни имена в C#?
а) *alfa-12*; б) *41beta*; в) *omega2delta*; г) *a_121*.
2. При следните декларации на променливи
`uint i; long j; short k; double x;`
кои от следните присвоявания са допустими:
а) `i = 2*j`; б) `j = k + x`; в) `k = i + 2`; г) `x = i + j + k`?
3. Каква ще бъде присвоена на променливата с стойност, ако стойността на *a* е 4, а на *b* е -3?
а) `a+b++`; б) `++a+b++`; в) `a%(-b)`; г) `a*b/2`.

```
int a = 5;
if (a > 0)
{ a = a + 5; }
else a = a - 5;
```

а.

```
int a = 5;
if (a > 0)
{ a = a + 5; }
a = a + 2;
```

б.

```
int a = 3, b = 3;
if (a == b )
{ a = a % b; }
else
{ if (b > 0)
{ a = a + b; }
}
```

в.

Фиг. 1.

4. След изпълнение на програмния фрагмент от *Фиг. 1а* стойността на променливата *a* е:
а) 5; б) 10; в) 0; г) друга.

5. След изпълнение на програмния фрагментот *Фиг. 1б* стойността на променливата *a* е:
 а) 10; б) 15; в) 17; г) друга.
6. След изпълнение на програмния фрагментот *Фиг. 1в* стойността на променливата *a* е:
 а) 10; б) 0; в) 12; г) 15.

```
int a = 1, b=2;
if (a > 0)
{ a = a + b; }
else
  a = b - a;
```

a.

```
int a = 1, b=2;
if (a < 0)
{ a = a + b; }
else
  if (b < 0)
  { a = b - a; }
```

б.

```
int a = 10, b=4;
if (a > 0)
  if (b < 0)
  { a = a * b; }
  else
  { a = a / b; }
```

в.

Фиг. 2.

7. След изпълнение на програмния фрагмент от *Фиг. 2а* стойността на променливата *a* е:
 а) 3; б) 1; в) 2; г) друга.
8. След изпълнение на програмния фрагмент от *Фиг. 2б* стойността на променливата *a* е:
 а) 1; б) 2; в) 3; г) друга.
9. След изпълнение на програмния фрагмент от *Фиг. 2в* стойността на променливата *a* е:
 а) 2; б) 40; в) 10; г) друга.

```
int a = 12, b=3;
if (a > 0)
  if (b > 0)
  { a = a % b; }
  else
  { a = a - b; }
```

a.

```
int a = 12, b=3;
if (a > 0)
  if (b > 0)
  { a = a % b; }
  else
  { a = a - b; }
```

б.

```
int a = 12, b=-3;
if (a <= 0)
{ a = a + b; }
else
{ if (b <= 0)
  { a = a % b; }
  else
  { a = a - b; }
}
```

в.

Фиг. 3.

10. След изпълнение на програмния фрагмент от *Фиг. 3а* стойността на променливата *a* е:
 а) 12; б) 4; в) 9; г) друга.
11. След изпълнение на програмния фрагмент от *Фиг. 3б* стойността на променливата *a* е:
 а) 12; б) 4; в) 9; г) друга;
12. След изпълнение на програмния фрагмент от *Фиг. 3в* стойността на променливата *a* е:
 а) 10; б) 0; в) 12; г) 15.

Речник

else	елз	иначе, в противен случай
event	ивѐнт	събитие
exerption	ексѐпшън	изключение, аварийно прекъсване на програма
if	иф	ако
read only	рийд онли	само за четене
size of	сайз ъф ...	размер на ...
unhandled	ънхѐндлид	необработено (за изключение)
update	ъпдѐйт	обновявам

Приложение Windows Forms Application

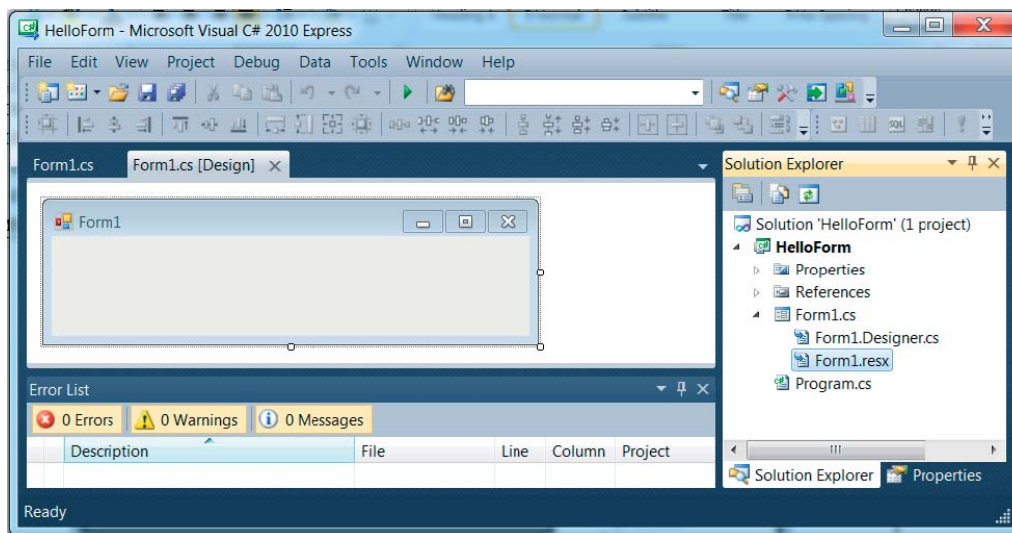
Вече знаем как се създават конзолни приложения. Време е да се научим да създаваме и приложения с графичен потребителски интерфейс, който днес се счита стандартен за приложните програми. Първите стъпки са подобни на тези, които правим при създаване на конзолно приложение. Стартираме средата и избираме New Project. В менюто за вида на приложението, обаче, посочваме Windows Forms Application. Както се вижда на *Фиг. 1* работното поле на средата вече изглежда по друг начин. Към познатите ни прозорци в списъка на Solution Explorer са се добавили файловете Form1.cs и Form1.cs [Design].

Както и при конзолните приложения, съхраняваме новосъздаваното решение под подходящо име, например HelloForm, в избрана от нас папка с командата File/Save All.

Графичен интерфейс

При стартирането на средата за създаване на приложение с графичен интерфейс, в работното поле се отваря прозорецът Form1.cs [Design]. В него е показан графичният елемент *екранна форма* (или просто *форма*), от който се създава основният прозорец на приложението (*Фиг. 1*). Формата е единствен обект на клас, с избрано от средата име Form1, който е *наследник* на класа Form. В ООП, когато един клас наследява друг, това означава, че той получава наготово всички методи на класа родител и в него може да се добавят липсващи в класа-родител възможности. Това е същността на програмирането на приложения с графичен интерфейс – трудните неща са направени в родителския клас, а за неопитния програмист остават да програмира нещо по-просто – функционалността която иска да реализира.

От класа-родител в класа Form1 се наследяват двата файла с имена Form1.cs и Form1.Designer.cs. В тях се съхраняват двата *изгледа* към създавания графичен интерфейс: за програмите и за дизайна, съответно. Дизайнерският изглед (прозорецът Form1.cs [Design], показан на *Фиг. 1*) е предназначен за създаване на графичния интерфейс. Върху изобразената в изгледа форма се разполагат необходи-

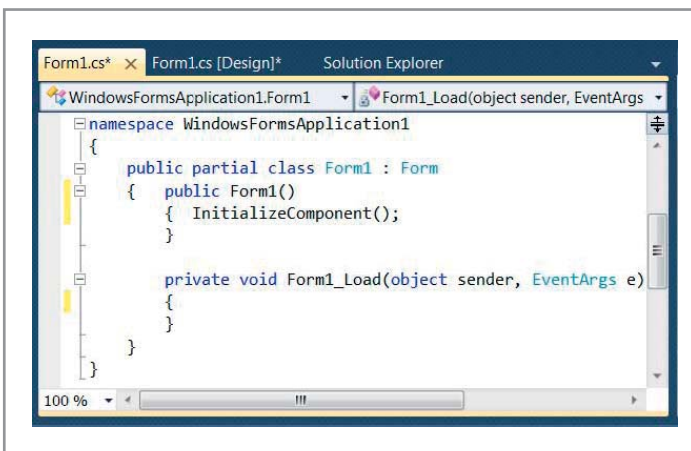


Фиг.1. Създаване на приложение с графичен интерфейс (Windows Forms Application)

мите за работата на приложението елементи на графичния интерфейс. Класът Form1 наследява и редица характеристики, наричани *атрибути* или *свойства*, с промяната на които можем да дадем на формата желани от нас вид.

Преминването от дизайнерския изглед в изгледа с програмите става с натискане на клавиша F7, а преминването в обратната посока – с клавишната комбинация Shift+F7. В програмния изглед (Фиг. 2) средата показва заготовка на програмния код, който реализира функционалността на приложението, където ще изписваме нужния ни код.

Още програмен код има във файловете Form1.Designer.cs и Program.cs. Това са програмните фрагменти, наследени от класа Form, които създават и извеждат формата при стартиране на приложението, създават и разполагат елементите във формата и т.н. Този код се създава и управлява от средата, обновява се автоматично, когато се добавят компоненти във формата и се променят свойствата им. Не е желателно да правите изменения в тези файлове засега.

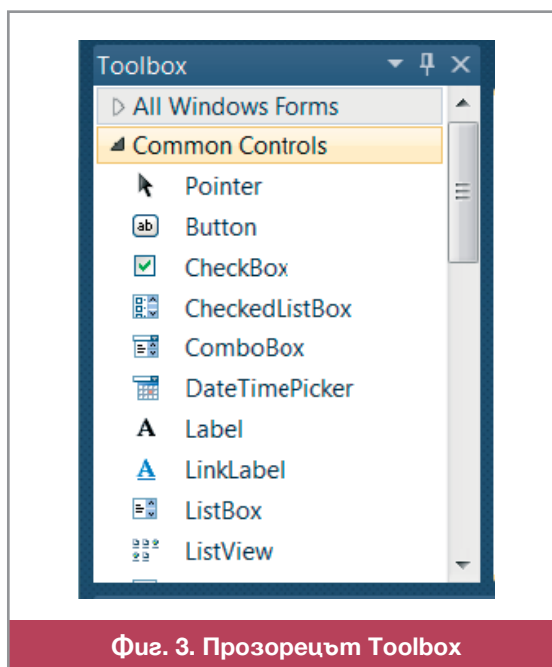


Фиг. 2.

Прозорецът Toolbox

Следващият елемент на средата, с който ще се запознаем е *кутията с елементи на графичния интерфейс* Toolbox (кутия с инструменти), които за кратко наричаме *компоненти* (Фиг. 3). Прозорецът Toolbox се отваря с едноименния бутон (🔧) или с командата View/Other Windows/Toolbox. По премълчаване средата го поставя неподвижен в лявата част на работното поле. Ако програмистът желае, може да го направи преместваем, като отвори менюто Windows Position с бутонна стрелка (📏), намиращ се в горната дясната част на прозореца и избере от него Float (плаващ).

Кутията Toolbox съдържа всички компоненти на графичния интерфейс, поддържани от езика C#. С част от тези компоненти и начинът, по който се вграждат в графичния интерфейс на приложните програми, ще се запознаем в следващите няколко урока. При визуалното програмиране най-естественият начин да се постави компонента в екранната форма е да се „хване“ с мишката съответната икона в прозореца Toolbox и да се „влачи“ до желаното място във формата.

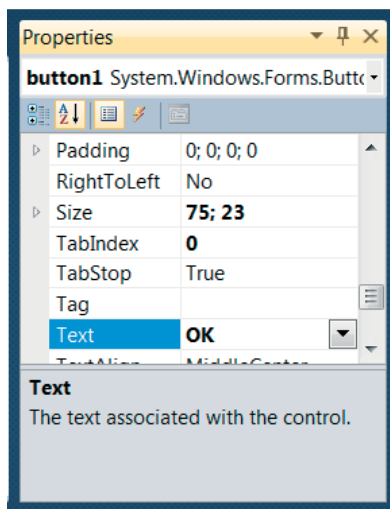


Фиг. 3. Прозорецът Toolbox

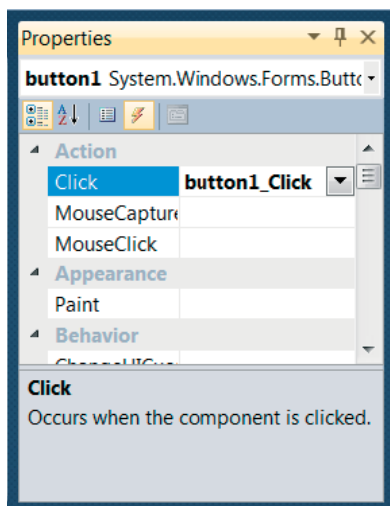
Прозорецът Properties

Последният елемент на средата, на който ще обърнем внимание в този урок, е прозорецът Properties (свойства), за който вече говорихме в началото на урока. Средата, обикновено, отваря този прозорец неподвижен, вдясно от основния, но при желание на програмиста и той може да бъде направен преместваем, също както прозорецът Toolbox. Да разгледаме по-подробно този важен за програмирането на графичен интерфейс прозорец.

Вече стана дума, че всички обекти на даден клас, включително екранната форма и компонентите, притежават различни свойства, с настройването на които програмистът изгражда интерфейса на при-



Фиг. 4. Свойства (в лексикографски ред)



Фиг. 5. Събития (по категории)

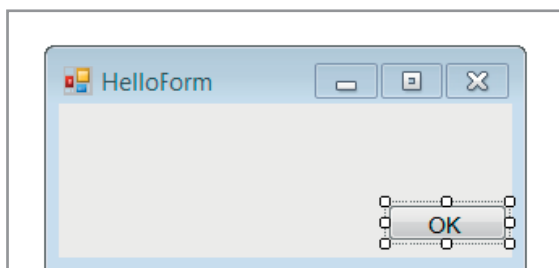
ложението. Особеност на класа Form и класовете от компоненти е, че освен свойства, с обектите от тези класове се свързват и *събития*, които могат да се случат по време на работата на програмата.

Всеки клас има специфични събития, но има и много събития които се срещат в повечето класове. Такива са, например, щракването с ляв бутон на мишката върху компонентата (Click) и смяната на съдържанието при компоненти в които може да се пише (TextChanged). Когато щракнем върху компонента, поставена във формата, можем да видим всичките ѝ свойства и събития в показания на Фиг. 4 прозорец Properties.

Под лентата с името на прозореца е разположена комбинирана текстова кутия, от списъка на която може да се избере обектът, чиито свойства и събития искаме да видим. Под тази кутия има лента с бутони. По премълчаване прозорецът Properties показва свойствата на обекта подредени в лексикографски ред на имената им (както са показани свойствата на Фиг. 4). За да видим свързаните с обекта събития, трябва да натиснем бутона Events (събития). Връщането към списъка със свойствата става с бутона Properties. Освен в лексикографски ред на имената, свойствата и събитията могат да бъдат показани групирани по категории (както са показани събитията на Фиг. 5), с натискане на бутона Categorized (категоризирани, групирани в категории). Връщането към лексикографска наредба става с бутона A-Z. Когато маркирате свойство или събитие в лентата, в долната част на прозореца се описва предназначението на избраното свойство или събитие.

За всяко свойство и събитие, в прозореца е отделен един ред с две полета. В лявото поле е изписано името на свойството или събитието. В дясното поле на свойство изписваме стойността му. В дясното поле на събитие, което искаме да обработваме в програмата, се изписва името на съответната функция, която обработва събитието.

Работа с компютър



Фиг. 6.

Стартирайте средата, отворете ново приложение с графичен интерфейс и го съхранете под името HelloForm, както е указано в началото на урока. След това изпълнете следните задачи:

Задача 1. От прозореца Properties променете свойството Text на формата в HelloForm, а размерите ѝ на 300,140.

Задача 2. От кутията с компоненти, с хващане и влачене, поставете в долния десен ъгъл на формата бутон. От прозореца Properties променете надписа на бутона (свойството Text) на ОК (Фиг. 6).

Задача 3. С двойно щракване върху бутона отворете прозореца с програмния код, където средата автоматично е добавила функцията `private void button1_Click()`, предназначена да обработи събитието Click – щракване върху бутона. Напише-

те в тялото на функцията оператора `Form.ActiveForm.Close()`; , който при щракване на бутона предизвиква затваряне на прозореца и с това прекратява изпълнението на програмата.

Компилирайте получената програма и я изпълнете. На екрана ще се отвори създаденият от програмата прозорец. Прекратете изпълнението, като щракнете върху бутона ОК.

Въпроси и задачи

1. Сравнете размерите в сантиметри на формата `HelloForm`, такива каквито се изобразяват на екрана при стартиране на програмата, с размерите зададени в свойството `Size` и определете колко екранни пиксела съответстват на 1 сантиметър от екрана на вашия компютър.
2. Навярно сте забелязали, че и в прозореца `Toolbox` има групиране на компонентите – `Common Controls` (компоненти с общо предназначение), `Containers` (контейнери; компоненти, които съдържат други компоненти), `Menus&Toolbars` и т.н. Групите в двата прозореца са оформени като папки, които се отварят и затварят, както папките във файловата система – с щракване върху триъгълния знак вляво. Отворете всяка от трите групи споменати по-горе и потърсете компоненти, които са ви познати от уроците по ИТ.

Упътване. Ако не разпознавате някоя компонента по иконата или името ѝ, включете я в екранна форма за да видите как изглежда, когато е в прозорец на програма.

Общи свойства

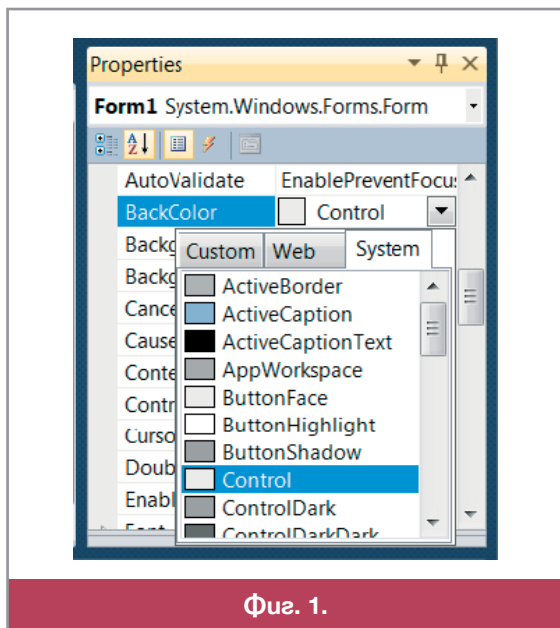
В този урок ще се запознаем с най-често използваните за създаване на графичен интерфейс компоненти, като посочим основните им свойства и събитията, които най-често свързваме с тях. Има няколко свойства, които са присъщи на всички компоненти, независимо от вида им. Затова да разгледаме първо тези, общи за всички компоненти свойства:

- ❖ **(Name)** – всяка компонента участваща в интерфейса на едно приложение има уникално *име*. Както видяхме при екранните форми това свойство винаги се показва най-горе в прозореца. За всяка включена във формата компонента, средата създава служебно име от името на класа, като променя първата буква от главна в малка и добавя уникален пореден номер. Например `button1`, `button2` и т.н. са компоненти от класа `Button`. Програмистът може да промени името, за да избере такова което подсказва презвазначението на компонентата.
- ❖ **Visible** – когато стойността на това свойството е `true`, компонентата е *видима*, т.е. изобразява се в прозореца, а ако е `false` – не се изобразява. Със задаване на желана стойност на това свойство по-време на изпълнение на програмата, програмистът може да променя динамично съдържанието на прозореца като скрива едни, а показва други компоненти.
- ❖ **Enabled** – промяната на това свойство от `true` във `false` оставя компонентата видима, но я прави *недостъпна*, т.е. функцията която тя изпълнява е блокирана за момента. Смяната от *достъпна* в *недостъпна* обикновено се придружава с видимо изменение на изображението на компонентата, например изрисуване в сиво.
- ❖ **Size** – *размер* на компонентата в пиксели. Свойство е *съставно*, композирано от две подсвойства – *ширина* (`Width`) и *височина* (`Height`) на компонентата. Пред имената на съставните свойства е поставен триъгълният знак, с който се показват или скриват подсвойствата. Стойностите могат да се редактират както в редовете `Width` и `Height` така и в обобщаващия ред `Size`.
- ❖ **Location** – това свойство също е съставно и съдържа координатите X и Y, в брой пиксели на горния ляв ъгъл на компонентата в съдържащия елемент – форма или друг контейнер. Оста *x* на екранната координатна система е насочена отляво надясно, а оста *y* – отгоре надолу.

Вече знаем, че всяко приложение с графичен интерфейс в ОС Windows трябва да има точно един *основен прозорец*, който се създава, като в обект от класа `Form` се поставят останалите елементи на главния прозорец на програмата. Празната екранна форма се създава автоматично от средата, само при отваряне на ново приложение с графичен интерфейс. Затова в прозореца `Toolbox` няма такава компонента.

По премълчаване средата дава на екранната форма името `Form1`, но без да променя началната буква на името на класа от голяма в малка. За тази компонента свойството `Location` на практика не съществува, тъй като няма елемент който да съдържа екранната форма на приложението. Мястото на основния прозорец на програмата върху екрана се задава в свойството `StartPosition`. По премълчаване средата дава на това свойство стойността `WindowsDefaultLocation`, т.е. там където ОС прецени за най-подходящо.


По-важни други свойства на формата са:



- ❖ `Text` е свойство от тип `string`. В него се задава заглавието на прозореца, което се изписва в лентата в горния му край. Заглавието най-често е името на програмата.

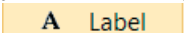
- ❖ `BackColor` е *цветът на фона* на формата, т.е. цветът в който е оцветена нейната вътрешност. Тъй като цветовете от които програмистът може да избира са много, дясното поле на свойството, в което трябва да се изпише избраният цвят, е комбинирана текстова кутия, в списъка на която са поставени възможните цветове (Фиг. 1). Такива комбинирани кутии има в десните полета на много свойства. Не само на тези, за които стойностите са много, но и на тези за които има само две възможности, например, `true` и `false`.

- ❖ `ForeColor` е цветът, с който по премълчаване се изписват текстовете на всички включени във формата компоненти. Ако в някоя от компонентите, в която има текст, се постави друг цвят в нейното свойство `ForeColor`, то изписването на текста в тази компонента ще стане с този цвят.

- ❖ `ControlBox` активира/деактивира групата от три бутона , които управляват показването на прозореца. Тези три бутона познаваме от уроците по ИТ. Припомнете си тяхното предназначение.

Останалите компоненти на прозореца на програмата се избират от прозореца `Toolbox` и се добавят във формата с хващане и влачване или като щракнем с левия бутон на мишката върху иконата на компонентата и след това върху избраното за компонентата място във формата. При необходимост, чрез хващане и влачване на компонентите или страните им, може да се променят тяхното местоположение или размер.


Emukem

Основното предназначение на компонентите от класа `Label` (етикет) е да се поставят надписи в основния прозорец на програмата и останалите контейнери. В етикети се поставят различни заглавия на прозорците и поясняващи надписи за предназначението на останалите компоненти. В етикети могат да се извеждат и стойности, които трябва само да се покажат в прозореца но не бива да се изменят от потребителя – например, резултати от извършените от програмата пресмятания. В прозореца `Toolbox` етикетът е представен с реда .

Текстът, който ще се изпише в етикета се задава в свойството му `Text`. При поставяне на етикета в екранната форма, по премълчаване съдържанието на свойството `Text` съвпада с името на етикета (Фиг.

2а). За да поставим искания от нас текст трябва да го изпишем в полето за редактиране на свойството (Фиг. 2б). Друго важно свойство на етикета е Font – съставно свойство, в което се задават параметрите на използвания шрифт, с много подсвойства. Най-важните от тях са Name и Size, задаващи вида и размера на шрифта, както и определящите стила – Bold, Italic и Underline (Фиг. 2в).


Текстова кутия

Компонентите от класа TextBox (текстова кутия) са предназначени за въвеждане на данни от клавиатурата по време на изпълнение на програмата. Точно това е и основното различие между текстовата кутия и етикета. В останалото, свойствата на тези две компоненти са почти идентични. В прозореца Toolbox текстовата кутия е представена с реда  TextBox.

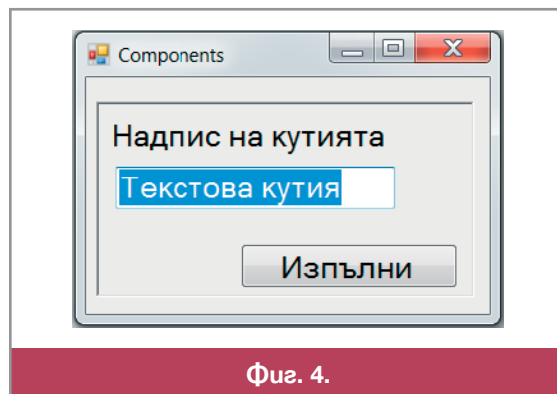
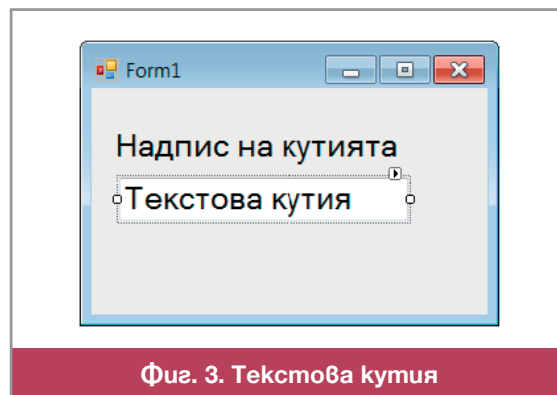
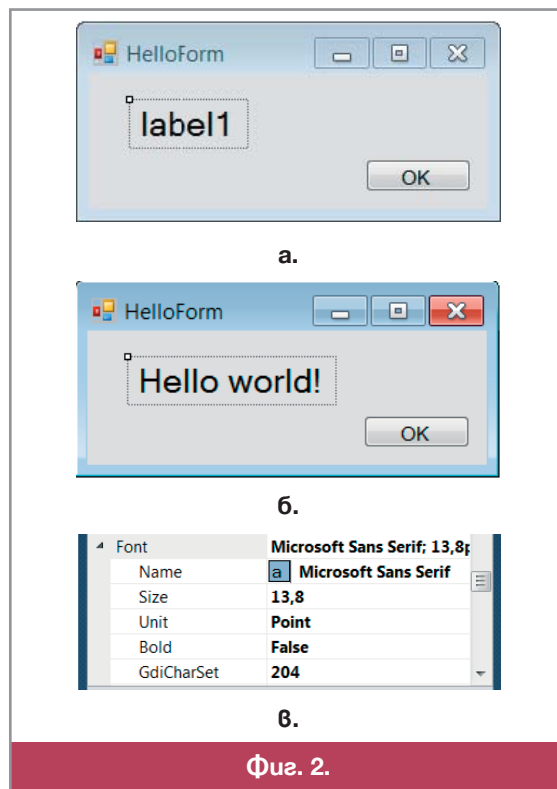
Компонентата позволява да се въвеждат данни на един или на много редове, което се управлява от свойството Multiline. След като потребителят е въвел данните в текстовата кутия, те стават достъпни в програмата като съдържание на свойството Text. Като съдържание на това свойството, обаче, по време на проектиране на формата или програмно, може да се постави текст който ще се покаже при отваряне на прозореца и може да бъде съдържание на кутията по премълчаване или подсказка за потребителя какво да въведе в полето (Фиг. 3).

Компонентите, в които потребителят може да въвежда данни, наричаме *активни*. Много е важно програмата да „научи“, че в съответната компонента потребителят е въвел данни. Затова, важно събитие за текстовата кутия е TextChanged (текстът е променен). Когато това събитие се случи програмата би трябвало да го обработи, като съхрани въведеното в кутията в съответна променлива.

Бутон

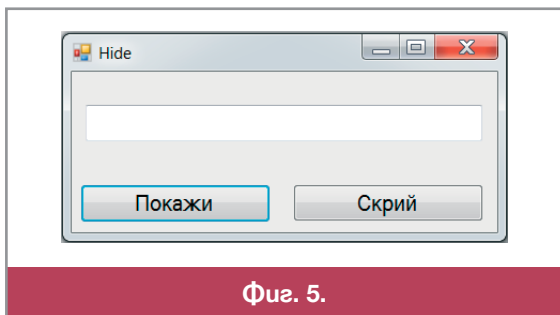
Компонентите от класа Button (*бутон*) са предназначени за подаване на команди от страна на потребителя към изпълняваната програма, затова са наричани още *командни бутони*. В прозореца Toolbox бутонът е представен с реда  Button. Основно свойство на командния бутон е надписът му, който е стойност на свойството Text и е добре да подсказва командата, която ще се стартира, когато потребителят щракне върху него с мишката (вж. бутон на Фиг. 4).

За тази компонента характерно е свойството Click, което се генерира в компютъра, когато потребителят натисне и след това бързо отпусне левия бутон на мишката. Затова средата автоматично генерира заготовка на метода, с който това събитие да бъде обработено, когато щракнем върху бутоната. В тази функция програмистът изписва програмен код, който изпълнява свързаната с бутоната команда.



Компонентите от класа **Panel** (панел) са контейнери, в които се поставят други компоненти, обединени от общо предназначение или близост на функциите им. Характерно за панела е свойство **BorderStyle**, стойността на което задава вида на рамката. Етикетът, текстовата кутия и бутонът на *Фиг. 4* са събрани в панел, рамката на който е оформена в 3D-стил.

Работа с компютър



Стартирайте средата, отворете ново приложение с графичен интерфейс и го съхранете под името **Hide**. Поставете в него етикет, текстово кутия и бутон, така че да получите прозореца, показан на *Фиг. 5*. Размерът на шрифта да е 14, а заглавието на формата да съвпада с името на програмата. Компилирайте програмата и проверете дали видът на прозореца отговаря на исканото.

Отворете от папка **Razdel_3** програмата с графичен потребителски интерфейс **Hide** (файлът **Hide.sln**). Формата ѝ съдържа текстово кутия и два бутона. Единият бутон е с надпис **Покажи**, а другият – с надпис **Скрий** (*Фиг. 5*).

При натискане на бутона **Покажи** в текстовата кутия трябва да се изпише низът **Hello World!**, а при натискане на бутон **Скрий**, текстовата кутия трябва да се скрива.

Разгледайте кода на това приложение:

```
namespace Hello_World
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            textBox1.Text = "Hello World!";
            textBox1.Visible = true;
        }
        private void button2_Click(object sender, EventArgs e)
        {
            textBox1.Visible = false;
        }
    }
}
```

Двете функции **button1_Click** и **button2_Click** са свързани със събитието **Click** на двата бутона. Заготовката на тези функции – заглавния ред с името, списъкът от параметри, които ние все още не използваме и двете къдрави скоби на тялото се създават автоматично, при двукратно щракване върху съответния бутон или при щракване върху събитието **Click** в прозореца **Properties**.

За програмиста остава по-трудната работа – да допише в заготовките програмните фрагменти, които изпълняват това, което поискахме да стане при „натискането“ на всеки от бутоните. При натискане на бутона с надпис **Покажи** (**button1**) трябва да се напише в текстовата кутия (**textBox1**) **Hello World!**:

```
textBox1.Text = "Hello World!"; // показва текст в кутията
```

и след това да направим кутията видима, защото при натискане на другия бутон тя се скрива:

```
textBox1.Visible = true; // прави кутията видима
```

При натискане на бутона с надпис **Скрий** (**button2**) трябва само да се направи текстовата кутия невидима:

```
textBox1.Visible = false; // прави кутията невидима
```

Компилирайте програмата и я изпълнете за да проверите работоспособността ѝ. **Забележка.** Макар и с друго предназначение, бутонът **F5** също ще стартира програмата.

Този урок ще посветим на разработване на неголямо приложение с графичен интерфейс. Тъй като това е първата ни самостоятелна програма с графичен интерфейс, ще използваме малко компоненти и проста функционалност.

Задача. Напишете програма с графичен интерфейс, която да позволява въвеждане на цифра и след натискане на бутон с надпис *С думи* да изписва въведената цифра с думи.

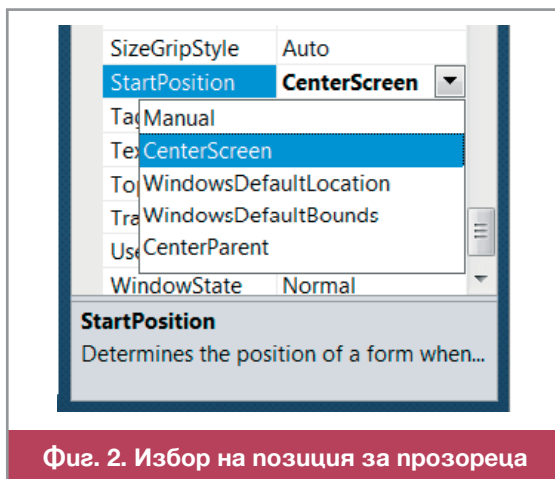
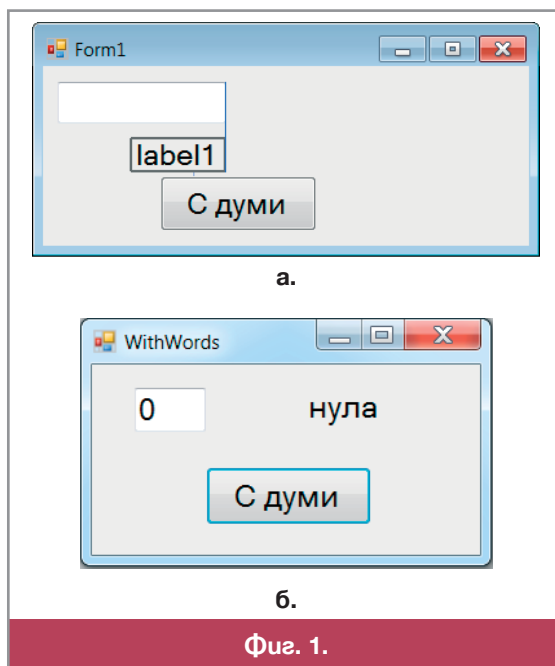
Проектиране на формата

От менюто *File/New Project* изберете *Windows Forms Application* и изберете име на проекта, на пример *WithWords*. Екранната форма се намира в прозореца *Form1.cs [Design]*. Може да промените размера ѝ по всяко време на проектирането, като хванете с мишката някоя от трите маркирани с бели квадратчета точки по страните на формата и влачите – надолу, надясно или по диагонала.

В екранната форма поставете текстова кутия за въвеждане на цифрата, един етикет, в който програмата ще изписва цифрата с думи и един бутон, с който потребителят да стартира изписването. Изберете необходимите компоненти от прозореца *Toolbox*. Оставете имената на трите компоненти такива, каквито средата ги е поставила – *textBox1*, *label1* и *button1*. С тези имена ще си служим в указанията по-долу. Разположете трите елемента така, че формата да е добре балансирана. Това не е лека задача. Чувството за добре проектирана и балансирана форма се изработва с много практика. На *Фиг. 1а* е показана една не добре балансирана форма – трите компоненти са струпани в лявата половина на формата, разстоянията между етикета и другите две компоненти са различни, текстовата кутия е твърде широка за единствената цифра, която ще се въвежда в нея. Формата вдясно на *Фиг. 1б* е доста по-добра. По време на местене на компонентите, върху формата се появяват сини линии (виж *Фиг. 1*). Използвайте тези линии за да подравнявате компонентите една спрямо друга – хоризонтално и вертикално.

На следващата стъпка от разработване на приложението можете да се заемете с надписите. Всички те се задават в прозореца *Properties*, в свойството *Text* на съответните компоненти. Променете свойството *Text* на формата така, както искате да се нарича програмата. Този надпис трябва да е достатъчно информативен, защото ще се показва в лентата на прозореца и по него потребителят разбира, за коя програма става дума. Текстовата кутия и етикетът може да оставите без надписи или пък да сложите цифрата *0* в текстовата кутия и низа *нула* – в етикета, за да подскажете какво прави програмата (*Фиг. 1б*).


Позицията на прозореца при стартиране на програмата задаваме в свойството *StartPosition* на формата. Можете да оставите стойността по премълчаване *WindowsDefaultLocation*, т.е. ОС да избере подходящо място, където да се отвори прозорецът при стартиране на програмата. Или да посочите своето предпочитание, например *CenterScreen*, т.е. в средата на екрана (*Фиг. 2*).



Компилирайте програмата и проверете дали при изпълнение формата има вида който сте очаквали. Затворете прозореца с бутона .

Програмиране на функционалността

Програмата е готова като интерфейс, но няма нужната функционалност – каквото и да въвеждаме в текстовата кутия, при натискане на бутона С думи съдържанието на етикета остава каквото е било при стартирането. За да получим функционалността която се иска в задачата, трябва да обработим събитието Click на бутона.

Активирайте прозореца със събитията на `button1` с  и щракнете двукратно в полето отдясно на събитието Click или направо щракнете двукратно върху самия бутон. Ще се отвори прозорецът за въвеждане на програмния код (`Form1.cs`). В него средата C# е включила автоматично заготовка на метода за обработване на събитието:

```
private void button1_Click(object sender, EventArgs e) { }
```

В тялото на метода трябва да напишете код, който да провери съдържанието на `textBox1` и да въведе подходящ текст в `label1`. Ако съдържанието на `textBox1` е 0, тогава текстът в етикета `label1` трябва да остане нула, иначе – ако цифрата е 1, текстът в етикета `label1` трябва да стане едно, ако цифрата е 2, текстът трябва да стане две, и т.н. Напишете тази проверка на езика C#. Вече знаете да си служите с оператора `if`, както и че името на свойство се изписва след името на обект, разделено от него с точка. Ето проверката за първите няколко цифри:

```
if (textBox1.Text == "0") label1.Text = "Нула";  
else if (textBox1.Text == "1") label1.Text = "Едно";  
else if (textBox1.Text == "2") label1.Text = "Две";  
else if (textBox1.Text == "3") label1.Text = "Три"; и т.н.
```

Допишете програмата, компилирайте я и я тествайте с всяка от цифрите, задавани в произволен ред. Какво става, ако се въведе число, което не е едноцифрено? В такъв случай би трябвало да се изведе някакво съобщение в `label1`. За целта допълнете последния `if` с `else` част, която да извежда текста:

```
else label1.Text = "Не знам!";
```




Компилирайте и проверете отново, въвеждайки двуцифрено число. За да обърне потребителят повече внимание на това **Не знам!**, може би си струва да го изпишем с удебелени букви. Смяната на шрифта става, като променим съдържанието на свойството `Font` на компонентата. Тъй като свойството е съставно, не можем да променим шрифта с просто присвояване. За целта трябва да се създаде нов обект от класа `Font`:

```
label1.Font = new Font(label1.Font, FontStyle.Bold);
```

Новият обект от класа `Font` ще бъде създаден от шрифта, който е бил зададен до момента в свойство `Font` на етикета `label1`, като стилът му се промени на удебелен (`FontStyle.Bold`). Стартирайте програмата и въведете в `textBox1` цифра. Програмата работи добре. Въведете двуцифрено число. В етикетът ще се изпише **Не знам!**, но вече с удебелен шрифт. Ако сега отново въведете цифра, името ѝ ще се изведе удебелено. Това е „бъг“ (грешка в програма)! Следователно, преди да проверява коя е цифрата в `textBox1`, програмата трябва да възстанови шрифта в нормален (`Regular`).

```
label1.Font = new Font(label1.Font, FontStyle.Regular);
```

Въпроси и задачи

1.  Напишете програма с графичен интерфейс, подобна на тази от урока, която въвежда в текстовата кутия двуцифрено число и извежда това число изписано с думи в етикета.
2.  Обединете програмата от урока с тази, която е решение на предната задача, така че потребителят да може да въвежда както едноцифрено, така и двуцифрено число.
- 3*.  Изписването на проверката за една от няколко възможности с вложени оператори `if...else` е досадно. Разгледайте възможностите на оператора `switch...case`, който прави същото, като вложените `if...else` (част от редовете са пропуснати):

```

switch (i) {
    case 1: label1.Text = "едно"; break;
    case 2: label1.Text = "две"; break;
    case 3: label1.Text = "три"; break;
    ...
    case 9: label1.Text = "девет"; break;
    default: label1.Text = "Не знам?";
             label1.Font = new Font(label1.Font, FontStyle.Bold);
             break;
}

```

Обяснете действието на оператора. Заменете вложените `if...else` оператори с оператор `switch...case`. Компилирайте и проверете работоспособността на програмата.

24
25

Проект: периметър и лице на фигури

Анализ на задачата


Припомнете си програмата `square`, която писахме в предишен урок. В нея въвеждахме дължината на страна на квадрат, а тя извеждаше в конзолата периметъра и лицето му. После имахме за задача да напишем подобна програма за правоъгълник. Сега ще реализираме един проект, който ще обедини двете програми в програма с графичен интерфейс, като добавим към двете фигури и триъгълник. Да изясним първо какво искаме да прави програмата, т.е. какво ще се въвежда и извежда. След това да определим какви компоненти ще ни трябват и, накрая, какъв ще е алгоритъмът на програмата.

Исходните данни за трите фигури са едни и същи, но входните данни, очевидно, ще са различни за всяка от тях. За пресмятане периметъра и лицето на квадрат ще ни трябва само едно число – дължината на страната, докато за правоъгълник – дължините на две съседни страни. За пресмятане периметъра на триъгълник, пък, ще ни трябват дължините на трите му страни. А това ни насочва към пресмятане на лицето му също по трите страни, за да не въвеждаме други данни.

Преди потребителят да започне да въвежда данни, трябва да посочи за коя фигурите ще се извършат пресмятанията. Тя трябва да е точно една! След което да въведе данните за нея и да натисне бутон Изчисли за да види резултата. После да избере друга фигура и т.н.

Радио-бутони

За да посочим коя е фигурата ще ни трябва подходяща компонента. При работата с приложни програми с графичен интерфейс сме се срещали с елемент, който е подходящ за целта – *група от радио-бутони*. Това са няколко бутона, като във всеки момент само един от тях е натиснат. Казваме още, че натиснатият радио-бутон е *активен*. Всеки от радио-бутоните на групата е изобразен като кръг, като в кръга на активния има точка.

В C# няма компонента група от радио-бутони, но има компонента *радио-бутон* (`RadioButton`), представена в прозореца Toolbox с реда  `RadioButton`. Създаването на група от радио-бутони и синхронизирането на работата им така, че във всеки момент точно един да бъде „натиснат“, е задача на програмиста. Най-важното свойство на радио-бутонът е `Checked`. То е от булев тип и показва дали бутонът е натиснат или не. Другото негово свойство, което ще използваме, е `Text`. Досещате се за какво служи то, нали?

Планиране на формата

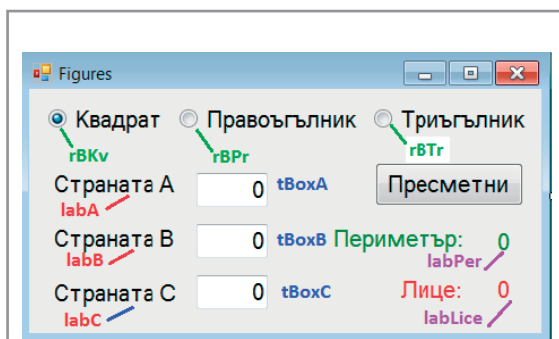
И така, във формата ще има три радио-бутона, обединени в група, надписани с Квадрат, Правоъгълник и Триъгълник, съответно. По подразбиране (при стартиране на програмата) ще е активен

бутонът Квадрат. Въвеждането на дължините на страните ще става в компоненти TextBox, като техният брой е различен за всяка фигура! Това означава, че когато е активен радиобутона Квадрат ще има една текстова кутия, за въвеждане на страната на квадрата. Когато, обаче, потребителят избере Триъгълник, във формата трябва да има три текстови кутии – по една за всяка страна. Затова при натискане на всеки от радиобутоните ще се обработва събитието му Click и ще се показват (<текстова кутия>.Visible = true) толкова текстови кутии, колкото трябва за съответната фигура, а другите ще се скриват (<текстова кутия>.Visible = false).

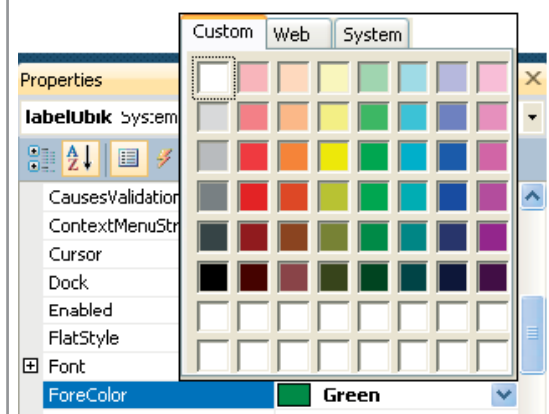
Работа с компютър

Отворете нов проект за създаване на приложение с графичен интерфейс и го съхранете под името Figures. Поставете във формата трите радиобутона, три текстови кутии с по един етикет за всяка от тях, един бутон за стартиране на пресмятанията и две двойки етикети за резултата. Във всяка от тези двойки единия етикет е за надписа – Периметър и Лице, съответно, а другият – за стойността на изчислените периметър и лице.

Поредете компонентите във формата така, както е показано на *Фиг. 1*. В свойството Text за всяко от тях въведете съответния текст. Не забравяйте да промените и свойството Text и на формата. Дайте на радиобутоните, етикетите (само тези, в които ще сменяме надписа или видимостта) и текстовите кутии имена (свойството Name) както е показано на *Фиг. 1*.









Фиг. 1.



Фиг. 2. Свойството ForeColor


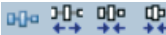



Както се вижда от *Фиг. 1*, за да подчертаем мястото където ще извеждаме резултата от пресмятанията, сме оцветили етикетите за показване на периметъра и лицето в друг цвят. Промяната на цвета на надписите става в свойството ForeColor (*Фиг. 2*). Когато щракнете в полето за стойност на свойството се отваря прозорец с три цветни палитри. Най-позната ни е тази която е в страницата Custom, защото сме я срещали в различни приложни програми, но не е трудно да се ориентирате как се използват и другите две палитри. Изберете цвета на текстовете с помощта на някоя от цветовете палитри.

При подреждане на формата, много е важно отделните елементи да са правилно подредени един спрямо друг. Например трите етикета с надписи за въвежданите страни да са подравнени точно един над друг вляво, както и разстоянието между първия и втория да е равно на разстоянието между втория и третия. Това можем да направим като изравним съответните стойности в свойството Location, но този начин е много бавен. За целта много по-добре е да се използват бутоните в инструменталната лента за дизайн на формата, показана на *Фиг. 3*.

За да приложи необходимата обработка, програмистът маркира компонентите за които ще се извърши операцията и натиска бутон. Така например, трите бутона най-вляво в лентата   , подравняват маркираните компоненти хоризонтално – вляво, центрирани и вдясно, съответно. Следващите три    подравняват вертикално. Следват, групата за изравняване на



Фиг. 3. Инструменти за дизайн на формата

размерите , групите за подравняване на разстоянията между компонентите – хоризонтално  и вертикално , и т.н. За упражнение маркирайте трите радио-бутона и ги подравнете вертикално нагоре , след което изравнете хоризонталните разстояния между тях (бутона ). Направете и други подравнявания, които ще подобрят вида на формата.

След като сте готови с дизайна на формата, време е да се заемете със същинската работа – програмиране на функционалността. Всъщност, тук е мястото да отбележим, че дизайнът на формата не е работа на програмистите и е по-добре да се извършва от съответните специалисти – компютърните дизайнери.

Функционалността на програмата ще се реализира с обработката на събитието Click на трите радиобутона и на бутона button1. Започнете с радиобутоните: маркирайте радио-бутона rBkV във формата, в прозореца Properties изберете Events и щракнете двукратно върху свойството Click. В прозореца с програмния код ще се покаже заготовката на метода за обработка private void rBkV_Click. Напишете в метода следния код:

```
private void rBkV_Click(object sender, EventArgs e)
{
    tBoxA.Visible = true; labA.Visible = true;
    tBoxB.Visible = false; labB.Visible = false;
    tBoxC.Visible = false; labC.Visible = false;
}
```

При натискане на радиобутона rBkV текстовата кутия tBoxA и етикетът labA ще станат видими, а другите две текстови кутии и техните етикети няма да се виждат. Напишете сами обработката на събитието Click за радиобутоните rBPr и rBTr. Използвайте копиране и вмъкване с последващо редактиране на вмъкнатия код, за да се справите по-бързо. Стартирайте програмата и пробвайте действието на радиобутоните.

За да получим окончателният вариант на програмата остана само да програмираме пресмятането на периметрите и лицата на трите фигури, което е тема на следващите два урока.

26
27

Проект: обработка на грешките

Преди да напишем код за обработката на събитието Click на бутона Пресметни, трябва да помислим какво точно ще има в програмния код, като ще потърсим ефективен алгоритъм.

Грешка, допускана от повечето начинаещи програмисти е, че след като разберат условието на задачата, веднага започват да пишат код. Така може да се подходи, ако алгоритъмът е тривиален и е сигурно, че програмата няма да се разширява след време. Нашата програма обаче е от тези, в които има какво да се допълни: да се добавят фигури – успоредник, ромб и др., или да се изчисляват и други елементи – диагонали, ъгли и т.н. Затова трябва много добре да се обмисли алгоритъмът ѝ. Колкото и неприятно да звучи, но в 90% от случаите първата хрумнала ни идея за алгоритъм не е най-добрата.

Какво всъщност трябва да направи програмата след натискане на бутона Пресметни? Да провери кой радио-бутон е натиснат, да превърне данните от текстовите кутии в числа, да изчисли периметъра и лицето и да покаже пресметнатите стойности във формата.

Първа идея за обработката на събитието Click на бутона Пресметни е показана на *Фиг. 1*. Веднага се забелязва повтарянето на действия – който и радио-бутон да бъде натиснат, ще трябва да се изпълни действието превърни низа от tBoxA в число и няма нужда това да се прави три пъти!

Вече имаме **втора идея** за алгоритъм. Действието превърни низа от tBoxA в число ще се извърши в началото

Ако активен е РБ rBkV
превърни низа от tBoxA в число
пресметни периметъра и лицето
иначе
ако активен е РБ rBPr
превърни низа от tBoxA в число
превърни низа от tBoxB в число
пресметни периметъра и лицето
иначе
ако активен е РБ rBTr
превърни низа от tBoxA в число
превърни низа от tBoxB в число
превърни низа от tBoxC в число
пресметни периметъра и лицето

Фиг. 1.

превърни низа от **tBoxA** в число

ако е вгугима tBoxB

превърни низа от **tBoxB** в число

ако е вгугима tBoxC

превърни низа от **tBoxC** в число

ако активен е РБ rBKv

пресметни периметъра и лицето

иначе

ако активен е РБ rBPr

пресметни периметъра и лицето

иначе

ако активен е РБ rBTr

пресметни периметъра и лицето

Фиг. 2.

и то безусловно. За да решим дали ще има превръщане на низ в число за всяка от другите две кутии трябва да проверим дали съответната кутия е видима в този момент (т.е. дали свойството `Visible` е `true`). Този алгоритъм е представен на Фиг. 2. На пръв поглед, той не изглежда много по-кратък, но в действителност ще спести доста програмен код, свързан с преобразуването на низ в число.

Защо превръщането на низа в число иска много код? Защото трябва да предпазим програмата от въвеждането на грешни данни. Спомнете си как, когато пишехме простия вариант на тази програма – пресмятащ само периметър и лице на квадрат – добавихме код с който елиминирахме случаите на въвеждане на отрицателните числа и нули. Ако потребителят въведе нецифрови знаци, обаче, програмата ще спре работа аварийно, което е

недопустимо. Сега ще се научим как да се справяме с подобни проблеми.

Оператор `try...catch...finally`

В езика `C#`, с оператора `try...catch...finally`, може да се обработват *изключения* (exception), т.е. ситуации, които ако не бъдат обработени, програмата ще спре аварийно.

Синтаксисът на операторът `try...catch...finally` е:

```
try { < оператори, при изпълнение на които очакваме изключение > }
catch (< вид на изключението >)
{ < оператори, които ще се изпълнят при възникване на изключение > }
finally { < оператори, които ще се изпълнят винаги > }
```

Когато програмата достигне този оператор, опитва да изпълни операторите в блока `try` (опитвам). В частта `catch...finally` операторът прилича на `if...else`, но разликата е значителна. Ролята на условие играе възникването на някакво очаквано изключение, като видът на изключението се поставя в скоби както условие. Ако изключението се случи, се изпълнява кодът след `catch` (хващам, улавям) а след това – кодът след `finally` (накрая). Ако очакваното изключение не се случи, се изпълнява само кодът след `finally`. Както частта `catch`, така и частта `finally`, може да липсва.

Работа с компютър

В нашия случай, например, няма нужда от частта `finally`:

```
try { a = int.Parse(tBoxA.Text); }
catch (FormatException)
{ a = 0; //при изключение а приема стойност 0
  tBoxA.Text = "0"; //записваме 0 и в текстовата кутия
}
```

Изключенията в `C#` са оформени в класове. Класът `FormatException` обхваща всички грешки, които възникват при опит за обработка на данни с неправилен формат. Такава грешка е, например, изписването на нецифрови знаци, там където се очакват цифри.

Кодът за обработка на събитието `Click` на бутона `Изчисли` е даден по-долу. В него няма нищо което да не познавате освен, може би, формулата за намиране на лице на триъгълник по 3 дадени страни: $S = \sqrt{p \cdot (p - a) \cdot (p - b) \cdot (p - c)}$, където a , b и c са дължините на страните, а p е полупериметърът:

$$p = \frac{a + b + c}{2}$$
 Тази формула се нарича *Херонова формула* за лице на триъгълник.

```
private void button1_Click(object sender, EventArgs e)
{ // Обработка въведеното в tBoxA
```


```

try { a = int.Parse(textBoxA.Text); }
catch (FormatException) { a = 0; textBoxA.Text = "0"; }
// Обработка въведеното в textBox
if (textBoxB.Visible)
{ try { b = int.Parse(textBoxB.Text); }
  catch (FormatException) { b = 0; textBoxB.Text = "0"; }
}
// Обработка въведеното в textBoxC
if (textBoxC.Visible)
{ try { c = int.Parse(textBoxC.Text); }
  catch (FormatException) { c = 0; textBoxC.Text = "0"; }
}
//Ако е избран радиобутон "Квадрат"
if (radioKv.Checked)
{ S = a * a; P = 4 * a; }
else //Ако е избран радиобутон "Правоъгълник"
  if (radioPr.Checked)
  { S = a * b; P = 2*(a+b); }
  else //Ако е избран радиобутон "Триъгълник"
  { P = a + b + c; p = P/2;
    S = Math.Sqrt(p * (p - a) * (p - b) * (p - c));
  }
//Показва намерените обиколка и лице
labPer.Text = P.ToString();
labLice.Text = String.Format("{0:0.###}", S);
}

```

Въведете този код за обработка на събитието Click на бутона Пресметни. Стартирайте програмата и я тествайте с различни входни данни.

Въпроси и задачи

1. Забелязахте ли някакви нередности при тестването на програмата Figures? Опишете ги и дайте предложение за отстраняването им.
2.  Променете програмата Figures така, че да може да приема като дължини на страните дробни числа.

28

Оформяне на проекта

Подпрограми

Програмата Figures, с която се занимавахме в предишните уроци е завършена. В края на предишния урок имаше въпрос, дали установихте някакви нередности по време на тестването? Една такава нередност е, че след като преминем от пресмятането за някоя фигура към пресмятане за друга фигура, в текстовите кутии остават стойностите от предното пресмятане! Това не е редно – когато потребителят избере с коя фигура ще работи, нормално е да види празни текстови кутии преди да започне да въвежда дължините на страните. Естествено, едната грешка води и до друга – след избиране на фигура от радиобутоните, в етикетите си стоят предишните изчисления на лицето и обиколката!


```
tBoxA.Text = "";
tBoxB.Text = "";
tBoxB.Text = "";
labPer.Text = "0";
labLice.Text = "0";
```

Фиг. 1.

```
void ClearBox()
{
    tBoxA.Text = "";
    tBoxB.Text = "";
    tBoxB.Text = "";
    labPer.Text = "0";
    labLice.Text = "0";
}
```

Фиг. 2.

Следователно, при обработване на събитието Click на всеки радиобутон, трябва да изчистваме текстовите кутии и двата етикета с резултата (Фиг. 1). Тези 5 реда може да напишем само веднъж и после да ги копираме и вмъкнем два пъти в другите два метода, които обработват събитието Click. Програмата започна да става дълга и трябва да се направи нещо!

Всяка система за програмиране дава възможност поредица от команди, които се повтарят в кода (дори едно повтаряне е излишно), да се оформят като *подпрограми*. За подпрограми говорихме в урока за историята на програмирането. Подпрограмата се пише веднъж и може да се използва наготово всеки път, когато главната програма трябва да изпълни кодираните в подпрограмата действия. В езиците произлезли от C подпрограмите се наричат *функции*. Вече знаем, че в обектно-ориентирано програмиране подпрограмите се наричат *методи*. Подпрограмите, несвързани пряко с обект, пък, се наричат *собствени методи* на класа в който са дефинирани.

Собственият метод се оформя както и всяка друга подпрограма – трябва да има уникално в рамките на класа име, за да може да се обърне програмата към него, последвано от списък с аргументи/параметри, по-

ставен в кръгли скоби. Преди името се пише задължително типа на резултата, който подпрограмата връща в извикалата я програма. Ако методът не връща стойност, тогава пред името се поставя ключовата дума **void**.

Изписването на името на метод в програма, последвано от знака точка и запетая, наричаме *извикване* на метода. При извикване на подпрограма, програмата прекратява изпълнението си, изпълнява се подпрограмата и след завършване работата ѝ, управлението се връща в извикващата програма, на мястото на извикването.

Работа с компютър

От програмния фрагмент на Фиг. 1 ще създадем метод ClearBox(), който да почиства формата при преминаване към нова фигура. Този метод няма да има аргументи и няма да връща никаква стойност. Кодът на метода е показан на Фиг. 2. Със създаването на собствени методи ще се запознаем по-подробно в един от следващите уроци!

В примера по-долу е показано къде трябва да се извика ClearBox () при обработка на събитието Click на радио-бутоната на квадрата:

```
private void RBKv_Click(object sender, EventArgs e)
{
    ClearBox();
    tBoxA.Visible = true; labA.Visible = true;
    tBoxB.Visible = false; labB.Visible = false;
    tBoxC.Visible = false; labC.Visible = false;
}
```

Обърнете внимание на кръглите скоби. Те задължително се пишат след името на метода независимо от това, че няма аргументи! Поставете извикване на ClearBox() при обработката на събитията Click на другите два радиобутона.

Стартирайте програмата. Сега, когато сменят фигурата, досадните остатъци от предишни въвеждания и пресмятания ги няма! Изглежда, че програмата е готова и може да я предоставите на ваш приятел, да си „поиграе“ с нея, без да знае, че в програмирането това се нарича *тестване*. Но ако той ви каже, че програмата е „супер“, значи не е добър „тестер“... Защото има още една възможност при която програмата работи неправилно за една от фигурите!

„Слабото място“ е триъгълникът. Знаем правилото, че три числа може да са дължини на страни на триъгълник, ако сумата на всеки две от тях е по-голяма от третото число. Ако въведете за дължини на страните 10, 2 и 2, за полупериметъра ще получите $p = (2+2+10)/2 = 7$ и тогава под корена във формулата на Хирон ще се получи $7.4.4.(-3) = -336$, число, което е отрицателно и не може да се коренува! Програмата ще завърши аварийно, като изведе в етикета за стойността на лицето знак (NuN), че не може да пресметне стой-

ност, която да изпише в етикета (Фиг. 3). Недопустимо! Ще се наложи да се напише още малко код...

Проверката за коректност на въведените дължини на страните на триъгълника може да се направи при обработка на събитието Click на бутона Пресметни. И само когато е изпълнено „правилото на трите страни“ – да се извърши пресмятането на лицето и периметъра. Какво да направите в противен случай? Има две възможности – да изведете нула или съобщение за грешка. За предпочитане е второто, защото иначе потребителят ще реши, че триъгълник със страни 2, 2 и 10 има лице нула, при положение, че такъв триъгълник въобще не съществува!

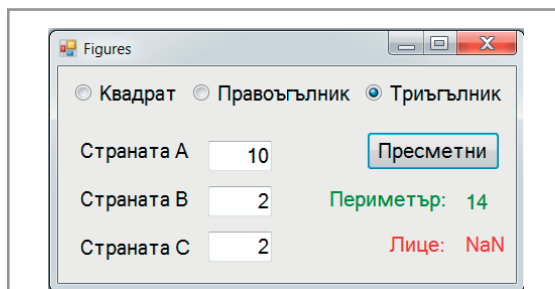
Проверката може да се реализира по много начини. Например, на променлива `f` от тип `bool` присвояват стойност `true`, т.е. предполагате, че входните данни са коректни. Проверявате за всеки две страни дали сумата от дължините им не е по-малка или равна на дължината на третата. Срещнете ли един такъв случай – сменяте стойността на булевата променлива на `false`. Ако след трите проверки стойността на променливата продължава да е `true` – извършвате изчислението, иначе трябва да изведете съобщение за неверни данни.

Извеждането на съобщението за грешни данни в етикет не е много удачно решение. Навярно сте забелязали как приложните програми в ОС Windows извеждат съобщенията за грешки в предназначен за целта прозорец. Ако вашата програма извежда съобщенията си в такъв прозорец, ще изглежда много по-професионално.

В C# това се осъществява с метода Show на класа MessageBox. Методът има само един аргумент – текстът на съобщението. Ето и кодът, който трябва да се постави там, където се изчисляват периметърът и лицето на триъгълника:



```
bool f = true;
if ((a + b) <= c) f = false;
if ((a + c) <= b) f = false;
if ((b + c) <= a) f = false;
if (f)
{
    P = a + b + c; p = P / 2;
    S = Math.Sqrt(p * (p - a) * (p - b) * (p - c));
}
else { MessageBox.Show("Въвели сте грешни страни!"); }
```

Въведете кода, компилирайте програмата и я тествайте. За да я направите още по-добра, след реда в който извеждате съобщение за грешка може да извикате отново процедурата ClearBox().



Фиг. 3.

Въпроси и задачи

1.  Допишете програмата така, че да отваря прозорец със съобщение Въвели сте некоректна стойност!, когато като дължина на страна се въвежда нула или отрицателно число.
2.  Допишете програмата така, че да може да пресмята периметър и лице на равнобедрен трапец по двете основи a , b и височина h .

Задача. Напишете програма, която да събира и извежда две цели числа. Числата ще бъдат въведени от потребителя в текстови кутии. Изборът на операция става с натискане на един от двата бутона, надписани с **+** и **-**. Програмата трябва да изведе в етикета label1 сбора или разликата на числата, в съответствие с натиснатия бутон. Ролята на бутона, надписан със **C** е, както в истинските калкулатори, да мо-



Фиг. 1.

же да се изчисти съдържанието на двете кутии и етикета, когато потребителят иска.

На Фиг. 1 е показан един възможен изглед на прозореца на програмата. Вие може да предпочетете друго пореждане на компонентите. Тъй като имате на разположение само един учебен час, няма да извършвате проверка дали потребителят е въвел букви вместо цифри в текстовите кутии.

Работа с компютър

Отворете ново приложение с графичен интерфейс и го озаглавете Calc. Поставете компонентите във формата, разположете ги добре в нея и променете размерите ѝ, ако е много малка или много голяма. Променете поставените от средата, по премълчаване, имена и надписи на компонентите така, че да напомнят за предназначението си.

Програмата ще трябва да обработи събитието Click на всеки от трите бутона. При надписаните с + и – бутони ще трябва да се вземат стойностите от текстовите кутии, да се извърши избраното аритметично действие и да се запише полученият резултат в етикета. При третия бутон – ще трябва да се изчисти съдържанието на текстовите кутии и етикета, като се изпишат в трите компоненти нули.

Напомняме, че за да отворите прозореца със заготовки за обработка на събитие, свързани с компонента, е достатъчно да щракнете двукратно върху компонентата във формата. Как се преобразува текст в число и обратно, може да вземете от написания в предишни уроци код, като смените имената на компонентите.

Успех!

30

Реализация на алгоритъм в метод

Подпрограми

В предишни уроци многократно беше споменавано понятието метод, използвахме наготово много от вградените в C# стандартни методи и дори написахме в предишен урок малък метод за почистване на текстови кутии и етикетите на екранна форма. В урока за историята на програмирането споменахме, за една от важните стъпки в развитието на технологията на програмирането е въвеждането на модулното програмиране, при което програмата се изгражда от отделни самостоятелни програмни модули, наричани най-общо *подпрограми*.

Подпрограмата е съставна част от програмата, която извършва определена обособена дейност (решава често срещана подзадача, извежда съобщения за състоянието на програмата в някакъв стандартен вид, реагира на събитие и т.н.). Когато в една програма се наложи да се изпълни действие, за което има написана подпрограма, тогава програмата *извиква* подпрограмата за да изпълни действието. Всяка подпрограма от своя страна може да извиква други подпрограми. След завършване работата на подпрограмата, *управлението* се предава обратно в извикващата програма. При това, подпрограмата може да *върне* в програмата някаква пресметната стойност или да не връща такава стойност.

Подпрограмата може да приема *параметри (аргументи)*, като по този начин настройва работата си и може да бъде доста разнообразна във функционирането в зависимост от получаваните параметри.

Различните езици наричат подпрограмите по различен начин. Езикът Pascal, например, разделя подпрограмите на *функции* и *процедури*, в зависимост от това дали пресмятат и връщат стойност или не. В езиците произлезли от C, подпрограмите се наричат *функции*.

Методу

В обектно-ориентираното програмиране подпрограмите се наричат *методи*. Всеки метод в C# трябва да бъде *член* на клас. Затова програмата на езика C# съдържа поне един клас, в който поставяме главния метод на програмата и нейна входна точка – методът *Main*.

Синтаксисът за дефиниране на метод е:

```
<тип> <име>(<параметри>)  
{ <оператори> [return <израз>;]  
}
```

Където *<тип>* е типът на връщаната от метода стойност – някой от познатите ни типове или `void`, когато методът не връща резултат. Изборът на *<име>* на метода става по общите правила за именуване в C#, като съществува традиция името да подсказва предназначението на метода, да започва с главна буква и да съдържа глагол (`Print`, `GetName`, `SetUserName` и др.) Списъкът с *<параметри>* (*аргументи*) може да е празен или да съдържа поредица от декларации на променливи. Променливите се декларират отделно, с тип и име, като се разделят една от друга със запетаи. Параметрите в дефиницията на метода наричаме **формални**.

Както се вижда от дефиницията, ако програмата връща някаква стойност от зададения *<тип>*, тогава в тялото на функцията непременно трябва да бъде включен поне един оператор `return <израз>;`, действието на който е да пресметне израза (стойността на израза трябва да е непременно от същия *<тип>*) и да прекрати изпълнението на подпрограмата като върне в извикващата програма пресметнатата стойност.

Списъкът с параметри на метода съдържа **само входните данни, които методът получава от извикващата програма**. Всички други променливи, които са необходими за запомняне на междинни резултати от пресмятанията или крайния резултат се декларират, по познатия ни начин, в тялото на метода (между отварящата и затварящата скоби). Такива променливи се наричат *локални* и имат *област на действие* (т.е. може да се използват) от мястото, където са декларирани, до края на метода. В един метод не може да се декларират две локални променливи с едно и също име! Дори списъкът с параметри да е празен, то двете скоби `()` винаги трябва да се изписват след името.

При работата над приложения с графичен интерфейс се наложи няколко пъти да напишем програмен код за тялото на методите, обработващи събития. Заготовките за тези методи, съдържащи главният ред на дефиницията и скобите на тялото се създават автоматично от средата и за програмиста остава да напише специфичния програмен код. Има, обаче, сериозни основания, налагащи създаване на собствени методи.

Ако в една програма се налага няколко пъти да се използва един и същ програмен фрагмент (или няколко фрагмента, разликата между които може да се преодолее с използване на параметри), тогава е задължително този фрагмент да се оформи като метод. Например, ако трябва да се почиства съдържанието на компоненти във форма след пресмятания, както в предишна програма, която написахме, или ако се наложи няколко пъти да намираме лицата на различни правоъгълници. Така се намалява дължината на програмата, защото няма да има повтаряне на код, програмата става по-лесна за четене и поддържане. Освен това, ако се окаже че във фрагмента има грешка ще я поправим само на едно място, а не навсякъде, където сме го използвали.

Примери

Ето например как изглежда метод за намиране лице на правоъгълник:

```
int GetRectangleArea(int width, int height)  
{ int area = width * height;  
  return area;  
}
```

Методът в този пример има два параметъра от тип `int` – ширината и височината на правоъгълника, а типът на връщаната от метода стойност също е `int`, защото ако дължините на страните на правоъгълника са цели числа, то лицето му също е цяло число. Името на метода е `GetRectangleArea`. Алгоритъмът пресмята лицето в променливата `area`, типът на която съвпада с типа на метода и връща тази стойност в извикващата програма с оператора `return area;`.

Важно е да се знае, че в тялото на един метод може да има няколко оператора `return`, но когато управлението достигне за пръв път до такъв оператор, изпълнението на метода се прекратява. Например:

```
int GetMaxOfThree(int a, int b, int c)  
{ if (a > b)  
  if (a > c) return a;
```

```

        else return c;
    else if (b > c) return b;
        else return c;
}

```

Методът `GetMaxOfThree` от този пример намира най-голямото от три цели числа и е очевидно че от трите оператора `return` ще се изпълни само този, в който е изписано името на променливата, съдържаща най-голямото от трите числа.

Последният ни пример:

```

void Print(double x)
{ Console.WriteLine("{0:###}", x);
}

```

е метод, който не връща резултат, а извежда в конзолата реалното число, зададено като параметър, в зададен формат. Така, ако в програмата се налага няколко пъти да се извежда реално число, ще се използва този метод и всички изведени числа ще бъдат форматирувани по един и същ начин.

Извикване на метод

Остана да разгледаме по-подробно как ще се извиква един метод от друг метод. За целта, на необходимото място, се изписва името на извиквания метод, последвано от списък с *фактически* параметри, заградени в кръгли скоби, и завършващ с точка и запетая. Важно е да се спазва правилото, че списъкът с фактическите параметри трябва да е със същата дължина, като списък с формалните параметри, и елементите с еднакви поредни номера в двата списъка трябва да са от един и същ тип.

Например, ако искаме да извикаме метода `GetRectangleArea`, за да намери лицето на правоъгълник със страни a и b , то трябва да напишем следното:

```
int area = GetRectangleArea(a, b);
```

Съвпадението на името на променливата на която метода присвоява стойност и името на променливата, в която методът пресмята стойността, в този пример, е случайност. Не е нужно тези две имена да съвпадат, но не е и забранено да съвпадат, тъй като са от различни *области на видимост* на променливите. Не е задължително също, съвпадането на имената на променливите от списъка на формалните параметри с имената на фактическите. Нещо повече, като фактически параметри може да се използват константи и дори изрази:

```
int max3 = GetMaxOfThree(a, 2*(x+y), 5);
```

В този пример името на един от фактическите параметри съвпада с името на съответния формален параметър в дефиницията на метода, един от фактическите параметри е константа, а третият – израз. Извикване на метод, който връща стойност, може да участва в израз и тогава, пресметнатата от метода стойност участва в пресмятането на израза там, където е извикването. Например:

```
int z = 2*GetMaxOfThree(a, 2*(x+y), 5) + a + 1;
```

Когато извикаме метод, който не връща резултат, т.е. типът му е `void`, тогава не е нужно да поставяме извикването в присвояване `Print(y)`;

Друга форма на извикване на метод, която често използваме, е, когато обект от класа в който е дефиниран методът трябва да бъде аргумент на метода. Тогава казваме, че методът се *прилага* към обекта: `<обект>. <име на метод> (<други параметри>)`. Такива са извикванията `Console.Write(<низ>)` и `Math.Sqrt(<дробна стойност>)`, например.

Въпроси и задачи

1. ■ Напишете метод за намиране на средно аритметично на три числа.
2. ■ Напишете програма, която намира сумата на целите положителни числа в интервала $[a;b]$. Двете променливи се въвеждат от клавиатурата, и са също цели и положителни числа.

Упътване: Напишете метод за пресмятане на формулата: $S = \frac{(a+b) \cdot (a-b+1)}{2}$, като направите проверка дали $a > b$?

3. ■ Напишете метод за решаване на уравнението $a \cdot x + b = 0$.

Упътване: Използвайте алгорътъма от урока Алгоритми. Методът не винаги ще връща число. Затова трябва да е от тип `string`, а числовите стойности да се преобразуват в този тип.

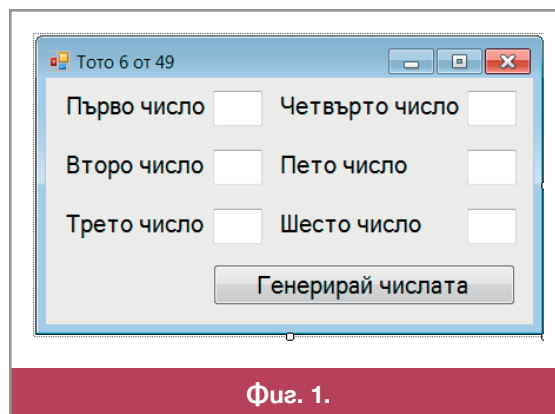
Задача. Да се напише програма, която при натискане на бутон в прозореца ѝ ще избира по случаен начин шест цели числа от 1 до 49 и ще ги показва в шест текстови кутии, т.е. ще симулира теглене на печелившите числа в тото играта „6 от 49“.

Дизайн на прозореца

Надяваме се, че изработването на дизайна на приложението (външния вид на прозореца му) вече не е сложна задача. Ще са необходими шест текстови кутии за числата, със съответните шест етикета за надписи и бутон за стартиране на генерирането (Фиг. 1).

Надпишете етикетите и бутона, както е показано на фигурата. Подредете компонентите във формата с използването на бутоните за подравняване. Не забравяйте да смените текста на прозореца с такъв който напомня за предназначението на програмата.

Подравнете етикетите и текстовите кутии хоризонтално и вертикално с бутоните за подравняване.



Фиг. 1.

Програмиране

Сега предстои да се напише програмния код. За целта, в метода за обработване на събитието Click на бутона трябва да се генерират шест случайни цели числа в интервала от 1 до 49, които да се запишат в шестте текстови кутии. Затова, напишете собствен метод, който да генерира случайно цяло число от 1 до 49. След това ще извикате този метод шест пъти, по веднъж за всяка от шесте кутии.

Методът, който ще напишете, ще връща цяло число, затова типът му ще е `int`. Не са необходими входни данни, т.е. списъкът с параметри на метода ще бъде празен. Нека името на метода е, например, `GetRandNumber`. В тялото на алгоритъма ще трябва да използвате един клас, който още не познавате – класът `Random`. Първо трябва да създадете обект от този клас.

Създаване на обект от зададен клас става с оператора

```
<име на класа> <име на обекта> = new <име на класа>();
```

където `<име на класа>()` е извикване на специален метод на класа, наричан **конструктор**. Името на конструктора трябва да съвпада с името на класа. Всеки клас има поне един конструктор. Ролята на конструкторите е да създават обекти от съответния клас, като отделните конструкции се различават по броя и типа на параметрите си и използват различни алгоритми за конструиране на обекта.

Конструирането на обект от класа `Random`, т.е. *генератор на случайни числа*, става с оператора `Random r = new Random();`.

Важен за генерирането на случайни числа е методът `Next()`, който приложен към обект от класа `Random` генерира по случаен начин цяло неотрицателно числа от типа `int`. За да ограничите интервала до `[1;49]`, използвайте операцията `%`. Остатъкът от деленето на едно неотрицателно число на 49 е цяло число от 0 до 48. Следователно, генерирането на нужното ни число ще стане с оператора:

```
int randNumber = r.Next() % 49 + 1;
```

Струва си да отбележим, че остатъкът на едно случайно число при делене е също случайно число. Изабщо, стойността на аритметичен израз, в който участва случайно число е случайно число!

Така, един възможен код на метода, който трябва да напишете е:

```
int GetRandNumber()
{
    Random r = new Random();
    return r.Next() % 49 + 1;
}
```

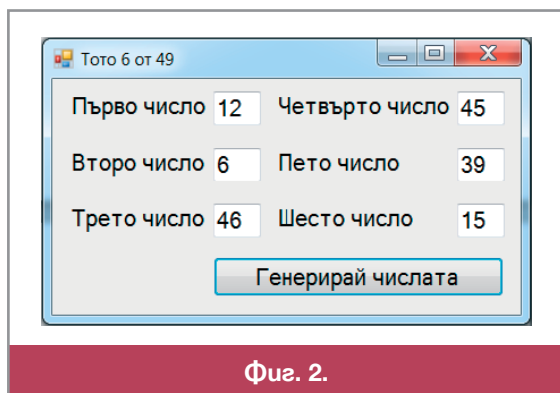
```
}
```

Вече може да напишете обработка на събитието Click на бутона. Започнете с генериране на първото число. Тъй като съдържанието на текстовата кутия е низ, трябва да превърнете генерираното число в низ с метода `ToString()` на класа `int`:

```
textBox1.Text = GetRandNumber().ToString();
```

Забележете как извикването на метода `GetRandNumber()` е поставено преди точката на прилагането на `ToString()` и така `ToString()` се прилага към генерираното от `GetRandNumber()` цяло число. Направете същото и за останалите 5 кутии. Ето как изглежда кодът, който трябва да поставите в тялото на класа `Form1`:

```
int GetRandNumber()
{
    Random r = new Random();
    return r.Next() % 49 + 1;
}
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Text = GetRandNumber().ToString();
    textBox2.Text = GetRandNumber().ToString();
    textBox3.Text = GetRandNumber().ToString();
    textBox4.Text = GetRandNumber().ToString();
    textBox5.Text = GetRandNumber().ToString();
    textBox6.Text = GetRandNumber().ToString();
}
}
```



Фиг. 2.

Стартирайте програмата и натиснете бутона, за да генерира числата. Забелязхте ли какво се случи? Програмата генерира едно и също число във всички кутии! Причината е в декларирането на обекта `r` от класа `Random` в тялото на `GetRandNumber()`. Така, при всяко извикване на метода, той създава отново обекта и започва генерирането с едно и също случайно число. Може да пробвате и ще видите, че при всяко стартиране на програмата, случайното число е различно (наистина „случайно“), но във всички кутии е едно и също. Когато ни трябва редица от различни случайни числа, създаването на генератора трябва да стане еднократно. Затова изнесете създаването на генератора извън метода за генериране на поредното случайно число, но в тялото на класа на формата. Тествайте програмата за да се уверите че грешката е поправена (Фиг. 2).

Следователно, може да се декларират променливи и обекти и извън всички методи на един клас, но в рамките на класа. Не е желателно, обаче, да го правите, освен ако няма основателна причина, както е в този случай.

Програмата вече работи, но ако я изпробвате няколко пъти може би ще забележите още един проблем – измежду случайно генерираните числа може да се появят еднакви. В тотализатора това е недопустимо, нали? Решаването на този проблем с материала, което познаваме до момента, ще доведе до много обемист и лошо структуриран код. Затова ще отложим решаването на проблема за следващ урок, когато ще сме изучили необходимия апарат.

32
33

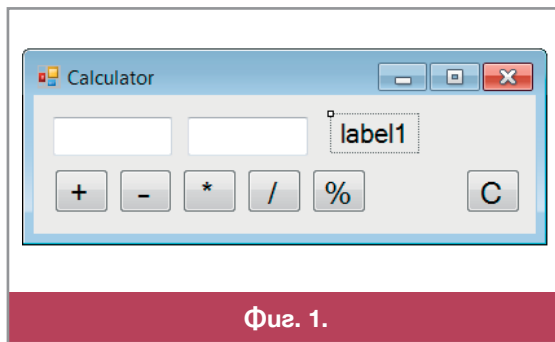


Проект: калкулатор

Задача. В този урок ще продължим работата по проекта, зададен в предишен урок за самостоятелна работа. Ще създадем отново калкулатор с графичен интерфейс, но с повече функции и стриктен контрол върху въвежданите данни. Калкулаторът ще е сравнително прост, само с аритметичните операции – събиране, изваждане, умножение, деление и намиране на остатък при деление. По-късно, ако желаете, може да разширите възможностите му и с други операции!

Проектиране на формата

Отворете ново приложение с графичен интерфейс и го съхранете под името Calculator. За проектиране на формата можете да използвате примера от предишен урок, в който за самостоятелна работа трябваше да направите калкулатор с две операции (Фиг. 1). Променете свойството Text на формата в Calculator. Добавете и три нови бутона с надписи *, / и %, за стартиране на операциите умножение, деление и намиране на остатък при деление. Подредете елементи така, че да са добре разположени по цялото поле на формата и ги подравнете с бутоните за подравняване.



Фиг. 1.

Програмиране на приложението

Ясно е, че методите за обработка на събитието Click на всеки от бутоните ще са еднотипни. Ще трябва да се вземат въведените в двете текстови кутии числа, зададени като низове, да се превърнат с метода `int.Parse()` в числа, като се контролира с оператора `try...catch...finally` наличието на неразрешени знаци, да се извърши операцията свързана с бутона и да се запише резултатът в етикета, като се превърне отново в низ с метода `ToString()` на класа `int`. И това трябва да се направи пет пъти, като всеки път променяме в кода само един знак на операция!

За да минимизираме повторението на код, ще напишем два собствени метода на класа `Form1`. Първият метод ще взема двата низа от текстовите кутии и ще ги превърща в числа. Вторият метод ще записва резултата в етикета. За метод, обработващ събитието Click на бутон на операция ще остане само да извика първия метод, за да получи операндите, да извърши операцията и да извика втория метод за да покаже резултата.

Да започнем с първия метод – той няма да връща стойност, а ще взема съдържанието на двете текстови кутии и ще опита да ги превърща в числа, с които после да се извършват операциите. Затова типът му ще бъде `void`. За да може, да се използват резултатите от него, две дробни числа ще трябва да бъдат декларирани извън всички методи на класа, за да бъдат достъпни във всеки метод. Методът ще се казва `GetNumbers` и списъкът му от параметри ще бъде празен:

```
double arg1, arg2, result;
void GetNumbers()
{ try { arg1 = double.Parse(textBox1.Text); }
  catch (FormatException) { arg1 = 0.0; textBox1.Text = "0"; }
  try { arg2 = double.Parse(textBox2.Text); }
  catch (FormatException) { arg2 = 0.0; textBox2.Text = "0"; }
}
```

Вторият метод ще записва резултата в етикета `label1`. Тъй като не връща стойност, типът му също ще е `void`, но ще има един параметър – пресметнатият резултат. Да наречем този метод `ShowResult`:

```
void ShowResult(double result)
{ label1.Text = result.ToString(); }
```

Вече може да напишем и методите, с които ще обработваме събитията Click на петте бутона. Да започнем с бутона за събиране:

```
private void button1_Click(object sender, EventArgs e)
{ GetNumbers(); ShowResult(arg1 + arg2); }
```

Напишете сами останалите четири метода. Компилирайте програмата и я тествайте с различни входни данни, включително и некоректни.

Тестване на програмите

Целта на тестването е чрез многократно изпълнение на програмата с различни входни данни да се открият евентуални програмни грешки. Не е шега подканянето в предишен урок да дадете готовата

програма на приятел „да си поиграе“. Програмистът на всяка цена трябва да извърши тестване на написаната от него програма, защото добре знае сложните за програмиране места в нея и може да избере тестови данни, с които да провери дали са правилно написани. Още по-добре е обаче тестването да се повтори от „независим“ тестер, например приятел, който ще направи това тестване добросъвестно. Независимият тестер не знае какво има в програмата и често измисля такива входни данни, за обработването на които програмистът дори не е и помислял. Затова, когато пишете по-сложни програми, намерете си и независим тестер.

В нашата програма, разбира се, все още има грешки. Например, ако въведем като втори аргумент числото 0 и опитаме да извършим операцията намиране на остатък при делене ще получим в етикета неразбираемия за средния ползвател надпис NuN. Малко по-добре се държи програмата при деление на 0 – извежда текста + безкрайност който, обаче, също може да уплаши някои потребители. Най-добре е, при въвеждане на втори аргумент 0 за тези две операции, да се отвори прозорец с подходящо съобщение и да не се извършва операцията, която математически е невъзможна:

```
private void button4_Click(object sender, EventArgs e)
{
    GetNumbers();
    if(arg2!=0) ShowResult(arg1 / arg2);
    else MessageBox.Show("Делене на нула!");
}
```

Поправете по същия начин и метода, който обработва събитието Click на бутона за намиране на остатъка при деление. Сега, ако се въведе 0 като втори аргумент, при натискане на тези два бутона няма да стане нищо нередно!

Въпроси и задачи

1. В екранната форма имахме бутон за почистване на двете текстови кутии и етикета. Напишете метод който извършва това и го извиквайте както при обработване на събитието Click на този бутон, така и след отказите за деление на 0.
2. Създайте приложение с графичен интерфейс, което решава уравнението $ax^2 + bx + c = 0$ за произволни дробни числа a , b и c .

Упътване. Направете блок-схема, подобна на тази за решаване на линейно уравнение, като включите всички възможни случаи за стойностите на коефициентите.

Речник

catch	кеч	улавям,хващам
categorized	кѐтегорйзд	категоризиран, класифициран в група
common	кѐман	общ
container	кѐнтѐйнър	съд, резервоар; контейнер,
event	ивѐнт	събитие
exception	иксѐпшн	изключение
finally	фѐйнъли	накрая, в заключение
float	флѐат	плавам, плаващ; обект, който си мени мястото
label	лѐйбъл	етикет
panel	пѐнл	панел
property	прѐпърти	свойство
toolbox	тѐулбокс	сандѐче с инструменти
try	трай	опитвам, пробвам

Цикли

От урока за алгоритми знаем, че една от най-често използваните конструкции в алгоритмиката е *цикличната*, при която няколко действия се повтарят многократно, докато е изпълнено някакво условие. Например, цикъл има в алгоритъма за събиране на числа в позиционна бройна система, като при всяка стъпка на цикъла трябва се съберат поредните две цифри и преносът, остатъкът от деленето на основата да се запомни като поредна цифра на сумата, а частното да стане пренос. Припомнете си и други алгоритмични процедури, които познавате и в които има повтарящи се стъпки.

Цикличната алгоритмична конструкция се реализира в програма чрез *оператори за цикъл*. Всеки език за процедурно програмиране притежава поне един такъв оператор. В езика C# има три оператора за цикъл – операторът `for`, операторът `while` и операторът `do...while`. В този урок ще се запознаем с един от тях – операторът за цикъл `for` (англ. за). Предназначението му е за създаване на цикли, в които тялото на цикъла трябва да се изпълни **за зададени стойности** на променлива или променливи.

Операторът за цикъл `for` има следния синтаксис:

```
for (<инициализация>; <условие>; <обновяване>)
{ <тяло на цикъла>
}
```

Операторът for

Изпълнението на оператора започва с изчисляване на израза *<инициализация>*. В него обикновено се задава начална стойност на променливата, която *управлява изпълнението* на цикъла. След това започва повтаряне на следните три действия:

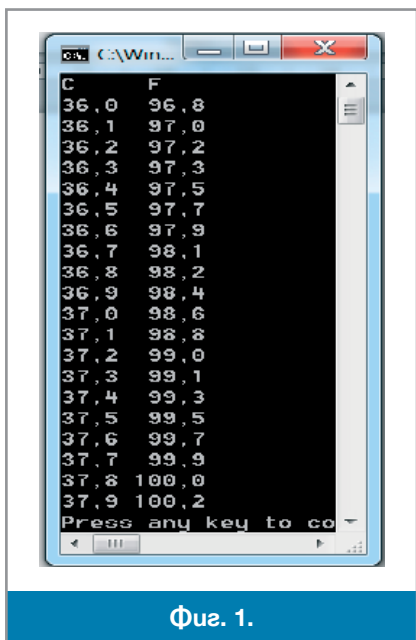
- ❖ изчислява се булевият израз *<условие>*. Ако стойността му е `false` – изпълнението на оператора за цикъл се прекратява. Ако стойността му е `true`, се продължава със следващите две стъпки;
- ❖ изпълнява се блокът от оператори, наричан *<тяло на цикъла>*;
- ❖ изчислява се изразът *<обновяване>*. В него, обикновено, се променя стойността на променливата, управляваща изпълнението на цикъла. След което отново се изчислява изразът условие и т.н.

Забележете, че трите израза са поставени в кръгли скоби, както условието на условния оператор, и са разделени един от друг с точка и запетая. Тялото на цикъла, както в условния оператор е блок от оператори, и когато съдържа повече от един оператор трябва да се поставя във фигурни скоби. Когато блокът се състои само от един оператор – тогава скобите не са зъдължителни, но може да се поставят за да се чете по лесно кодът.

 Работа с компютър

За пример, нека съставим таблица за преминаване от температурната скала на Целзий, към температурната скала на Фаренхайт. Формулата за преизчисляване на температурата е $f = 9/5 \cdot c + 32$, където c е температурата по скалата на Целзий, а f – съответната ѝ температура по Фаренхайт. В таблицата ще включим температури от 36°C до 38°C, през една десета от градуса.

Дробната променлива c , в която ще получаваме поредната температура по Целзий, ще бъде управляваща на цикъла. Стойността, с която всеки път променяме управляващата променлива, наричаме *стъпка на цикъла*. Ще ни е необходима и дробна променлива f , за температурата по Фаренхайт. На променливата c ще даваме последователно стойностите 36.0, 36.1, 36.2 и т.н. докато стигнем до 38.0.



Фиг. 1.

и за всяка стойност на *c* ще пресмятаме съответната стойност на *f* по формулата. За целта ще съставим цикъл `for`, в израза-инициализация на който ще дадем на *c* началната стойност: $c = 36.0$. Вторият израз ще бъде условието, при което цикълът да продължи да се изпълнява: $c \leq 38.0$, а в израза-обновяване ще увеличаваме всеки път стойността на управляващата променлива със стъпката: $c = c + 0.1$. В тялото на цикъла ще пресмятаме в *f* температурата по Фаренхайт и ще извеждаме двете стойности в таблицата:

```
static void Main(string[] args)
{
    double c, f;
    Console.WriteLine("C    F");
    for (c = 36.0; c <= 38.0; c = c + 0.1)
    {
        f = (9.0 / 5.0) * c + 32.0;
        Console.Write("{0,4:##.0}", c);
        Console.Write("{0,5:##.0}\n", f);
    }
}
```

Напишете конзолно приложение `CellFar` с тази главна функция.

Компилирайте го и го изпълнете. Резултатът трябва да е този, показан на Фиг. 1.

Проверете какво ще стане с резултата, ако вместо форматиращите елементи `{0,4:##.0}` и `{1,5:##.0}` напишете `{0,4:##.#}` и `{1,5:##.#}`. Опитайте се да обясните разликата в използване на `#` и `0`, при задаване броя на знаците след десетичната точка на извежданото дробно число.

Цикли с брояч

Много характерни за програмирането са циклите `for` управлявани с променлива *брояч*. Броячът е променлива от цял тип, на която се задават съответна начална и крайна стойност, като стъпката обикновено е единица. За примерите използваме променлива *i* от тип `int`, но това не е задължително. Декларирането на променливата-брояч, може да стане в оператора `for`, или предварително. Броенето може да се извърши *напред* с начална стойност по-малка или равна на крайната:

```
for(int i = <начална стойност>; i <= <крайна стойност>; i++)
{ <тяло на цикъла> }
```

или *назад*, с начална стойност по-голяма или равна на крайната:

```
for(int i = <начална стойност>; i >= <крайна стойност>; i--)
{ <тяло на цикъла> }
```

Например, цикълът

```
for(int i=1; i<=10; i++) { Console.WriteLine(i); }
```

ще изведе на конзолата числата от 1 до 10, всяко – в началото на нов ред, а цикълът

```
for(int i=10; i>=10; i--) { Console.WriteLine(i); }
```

ще изведе по същия начин числата от 10 до 1. Програмният фрагмент

```
int sum = 0;
for(int i=1; i<=10; i++) sum = sum + i;
```

пък, пресмята в променливата *sum* сумата на числата от 1 до 10. Забележете, как преди да започне сумирането в тялото на цикъла, сме инициализирали променливата, в която ще се натрупва сумата с нула и как сме използвали променливата брояч в тялото на цикъла, защото тя винаги съдържа поредното число, което събираме. Забележете освен това, че когато тялото на цикъла е един много къс оператор, можем спокойно да го поставим в реда на ключовата дума `for` и да не го поставяме във фигурните скоби.

Други възможности

Всъщност разгледайте до тук възможности на оператора `for` са много специални случаи. Както се вижда от дефиницията на оператора, трите израза може да са най-различни и с един оператор могат да се извършат много работи. Следващият пример е цикъл, в който се инициализират и променят на

всяка стъпка две променливи. Правилото е, че на мястото на всеки от изразите може да се напишат няколко израза, разделени със запетаи:

```
for (int i = 1, sum = 1; i <= 16; i = i * 2, sum = sum + i)
    Console.WriteLine("i={0}, sum={1}", i, sum);
```

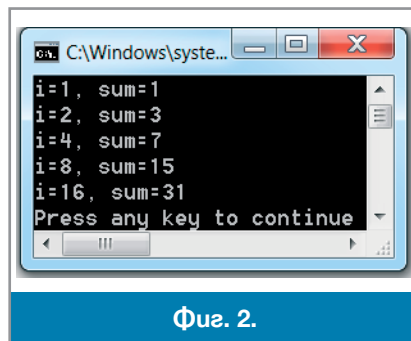
Резултатът от изпълнението на цикъла е показан на *Фиг. 2*.

В този пример и двете променливи растат, но е възможно едната променлива да расте, а другата – да намалява. Всъщност тялото на цикъла `for` може да е празно. Така например, цикълът, който намира сумата на числата от 1 до 10, може да се изпише и без тяло:

```
for (int sum=0, i = 1; i <= 10; sum = sum + i, i++);
```

Нещо повече, нито един от изразите в оператора не е задължителен – инициализацията може да се направи вън от оператора за цикъл, а проверката на условието и обновяването на управляващите променливи – в тялото на цикъла. В такъв случай, изпълнението на цикъла трябва да се прекрати явно с оператора `break`;

```
int i = 1, sum = 0;
for (;;) { sum = sum + i; i++; if (i > 10) break; }
```



Фиг. 2.

Работа с компютър

Задача 1. Напишете конзолно приложение `Maxval`, което намира най-голямото от зададени цели числа. Програмата първо трябва да въведе от клавиатурата ред с броя n на числата, $n > 0$, а след това n реда, с по едно от числата на всеки от тях. На конзолата програмата трябва да изведе най-голямото от n -те числа.

ПРИМЕР:	Вход:	Изход:
	4	23
	6	
	12	
	5	
	23	

Решение: Ще намерим в променливата `maxN` най-голямото от въведените числа. Ще инициализираме тази променлива с най-малката стойност на типа `int`, която е зададена в атрибута `MinValue` на класа `int`. След това ще въведем n и ще направим цикъл с брояч от 1 до n , който въвежда зададените числа и за всяко число проверява дали не е по-голямо от намерения до момента в `maxN` максимум. Ако това е така, ще заменяме съдържанието на `maxN` с новото по-голямо число. Програмата е показана на *Фиг. 3а*.

```
static void Main(string[] args)
{
    int n; string s;
    s = Console.ReadLine();
    n = int.Parse(s);
    int maxN=int.MinValue;
    for (int i = 1; i <= n; i++)
    {
        s = Console.ReadLine();
        int number = int.Parse(s);
        if (number > maxN)
            maxN = number;
    }
    Console.WriteLine("Max:{0}",maxN);
}
```

a

```
static void Main(string[] args)
{
    int n; string s;
    s = Console.ReadLine();
    n = int.Parse(s);
    int sum=0;
    for (int i = 1; i <= n; i++)
    {
        s = Console.ReadLine();
        int number = int.Parse(s);
        sum = sum + number;
    }
    Console.WriteLine("Sum={0}", sum);
}
```

b

Фиг. 3.

Задача 2. Направете необходимите промени в програмата `Maxval` така, че да получите програмата `Maxval`, която намира най-малкото от зададените числа?

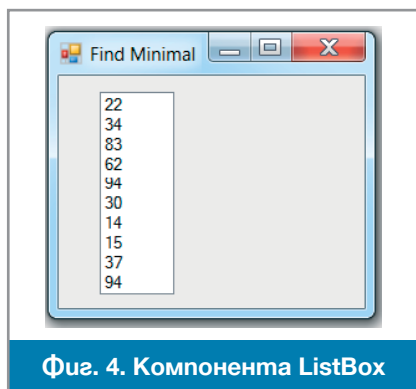
Задача 3. Напишете конзолно приложение `Sumval`, което да намира сумата на зададени цели числа. Програмата първо трябва да въведе от клавиатурата ред с броя n на числата, $n > 0$, а след това n реда, с по едно от числата всеки от тях. На конзолата програмата трябва да изведе сумата n -те числа.

ПРИМЕР:

Вход:	Изход:
5	62
6	
12	
5	
23	
16	

Решение: Също както в **Задача 1** трябва да организираме цикъл, който въвежда n -те числа и добавя всяко от числата в променливата за натрупване на сумата, която предварително е инициализирана с нула (Фиг. 3б)!

Компонента `ListBox`



След като вече знаем какво е циклична конструкция в алгоритъм и предназначението на оператора `for`, ще разгледаме още един пример за употреба на този оператор, този път в приложение с графичен интерфейс. За целта ще се запознаем с един нов елемент на графичния интерфейс – *списъчната кутия* (`ListBox`, `ListBox`). Тази компонента служи за показване на списък от елементи (числа, текст и др.) върху екрана (Фиг. 4). Елементите на списъка се съхраняват в колекцията `Items` (англ. елементи) на класа `ListBox`. За следващото упражнение ще ни трябват два от методите на колекцията:

- ❖ методът `Add(<елемент>)` добавя зададеният като аргумент елемент в списъка на кутията;
- ❖ методът `Clear()` е без аргументи и изтрива всички добавени до момента елементи в списъка, ако има такива. В резултат списъкът на кутията ще бъде празен.

Работа с компютър

Задачата е да напишем програма, която генерира десет случайни числа в интервала от 1 до 100, показва ги в списъчна кутия, след което намира минималното измежду генерираните числа. Потребителят трябва да може, с натискане на съответен бутон, да генерира многократно числа и, с натискане на друг бутон да намира минималното измежду генерираните числа.

Да започнем със създаването на списъчната кутия и генериране на числата за списъка. Създайте празно приложение с графичен интерфейс с име `FindMin` като от менюто `File/ New Project` изберете `Windows Forms Application`. Във формата, от която ще се създаде прозореца на приложението, добавете една списъчна кутия `listBox1` и един бутон `button1`, надписан с текста `Генериране`.

В метода `button1_Click` ще поставим код, който ще извърши генерирането на десетте числа и добавянето им в списъка на кутията `listBox1`. Заготовката за метода е в страницата `Form1.cs`, която се отваря с двукратно щракване върху бутона. За генериране на случайно число използвайте класа `Random`. Създайте обект `r` от класа `Random`, за да генерирате чрез него случайни числа:

```
Random r = new Random();
```

Сега можете да извикате 10 пъти метода `Next()` на обекта `r`, за да създаде 10 случайни числа, но очевидно по-бързо и по-лесно за изписване е, да използвате цикъл с брояч от 1 до 10. В тялото на този цикъл трябва да включите две действия, които да се изпълняват на всяка стъпка – генериране на случайно число от 1 до 100 и добавяне на това число в колекцията `Items` на списъчната кутия `listBox1`. За добавянето ще използваме метода `Add (<елемент>)` на колекцията. Така получавате кода:

```
private void button1_Click(object sender, EventArgs e)
{
    Random r = new Random();
    for (int i = 1; i <= 10; i++)
    {
        int number = r.Next()%100+1;
        listBox1.Items.Add(number);
    }
}
```

Поставете този код в програмата, компилирайте я и проверете работоспособността ѝ (Фиг. 5). Ако списъчната кутия не е достатъчно висока за да събере десетте числа, влезте в страницата Form1.cs [Design], и с влачване на мишката увеличете височината на контролата доколкото е необходимо, отново компилирайте и проверете дали кутията е достатъчно висока.

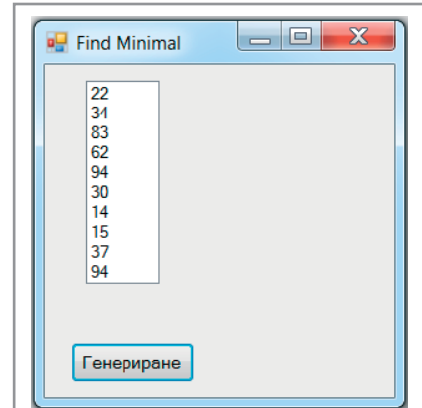
Сигурно сте забелязали, че ако натиснете бутона Генериране повторно, то новогенерираните числа се добавят след предишните. Това, разбира се, е нежелателно и трябва да бъде направено съответно изменение в кода. Всеки път, когато потребителят натисне бутона Генериране, съдържанието на списъчната кутия трябва да се изчисти. За целта, използвайте метода Clear() на колекцията Items. Извикайте метода: listBox1.Items.Clear(); преди началото на цикъла в който се генерират числата. Компилирайте и проверете работоспособността на новата версия.

Следващата стъпка е, да се добави една текстова кутия и един бутон. При щракване върху този бутон, програмата трябва да намери и покаже в текстовата кутия най-малкото от случайно генерираните числа. Влезте в страницата Form1.cs [Design] и добавете във формата бутон с име button2, надписан с текста Минимален елемент и текстова кутия textBox1.

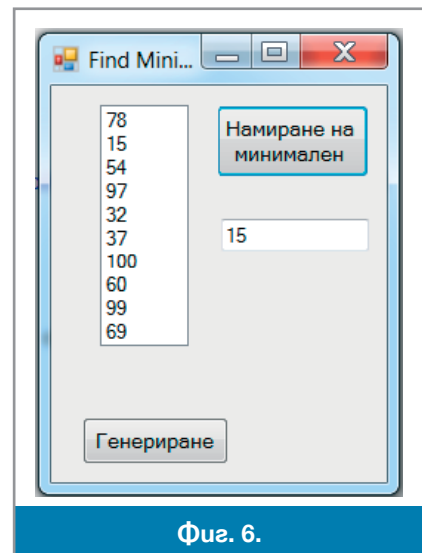
Най-добре е намирането на най-малкото число да стане още по време на генерирането. Както вече постъпихте в предишен урок, декларирайте променлива minN в рамките на класа Form1, а не на метода на button1, за да е видима и в метода button2_Click. В тази променлива, сравнявайки съдържанието ѝ с всяко от генерираните числа ще намерите най-малкото от тях. Променливата трябва да инициализирате преди началото на цикъла с най-голямото число от тип int – int.MaxValue:

```
int minN;
private void button1_Click(object sender,
EventArgs e)
{
    Random r = new Random();
    listBox1.Items.Clear();
    minN = int.MaxValue;
    for (int i = 1; i <= 10; i++)
    {
        int number = r.Next()%100+1;
        listBox1.Items.Add(number);
        if (number < minN)
            minN = number;
    }
}
```

Сега, вече може да напишете код за метода button2_Click(), в който на свойството Text на текстовата кутия textBox1 да се присвоява стойността на променливата minN. Тъй като свойството Text е от тип string, а стойността на minN е от тип int, то при присвояването, числовата стойност трябва да се преобразува в низ с метода ToString() на класа int.



Фиг. 5.



Фиг. 6.

```
private void button2_Click(object sender, EventArgs e)
{ textBox1.Text = minN.ToString();
}
}
```

Програмата е готова. Компилирайте я и проверете нейната работоспособност (Фиг. 6).

Въпроси и задачи

- Какво ще изведе в конзолата всеки от следните програмни фрагменти?
 - `for(int i=1; i<=10; i=i+2)`
`Console.WriteLine(i);`
 - `for(int i = 1, x = 20; i <= 128; i = i * 2, x--)`
`{ Console.WriteLine("i={0}, x={1}", i, x); }`
- Напишете програма, която въвежда от клавиатурата цяло положително число и проверява дали е просто, като извежда в конзолата подходящо съобщение. **Упътване.** Едно число е просто, ако се дели без остатък само на 1 и на себе си.
- Напишете програма, която да извежда на конзолата степените на числото 2 от 2^0 до 2^{49} . **Упътване.** Използвайте метода `Math.Pow()`.

35 Оператори while и do...while

Вече се запознахме с оператора за цикъл `for`. Той е подходящ за случаите, в които тялото на цикъла трябва да се изпълни **за определени стойности** на някаква управляваща променлива или **определен брой пъти**. За случаи, различни от споменатите, езикът C# предоставя две други възможности.

Синтаксисът на оператора `while` е:

```
while (<условие>)
{ <тяло на цикъла> }
```

Оператор while

Операторът `while` (докато) проверява стойността на *условието* и ако тя е `true`, тогава изпълнява *тялото на цикъла*. Операторът продължава да прави това, **докато** условието е изпълнено. Когато за пръв път стойността на израза стане `false`, операторът прекратява работа и предава управлението на следващия оператор. Тъй като в този оператор условието се проверява преди да се изпълни тялото на цикъла, за него казваме че е с *предусловие*.

<code>for (<инициализация>; <условие>; <обновяване>)</code>	<code><инициализация>;</code>
<code>{</code>	<code>while (<условие>)</code>
<code> <тяло на цикъла></code>	<code>{ <тяло на цикъла></code>
<code>}</code>	<code> <обновяване></code>
	<code>}</code>

Фиг. 1.

Всъщност възможностите на оператора `while` не са различни от тези на оператора `for`. На Фиг. 1 вляво е показан схематично оператор `for`, а в дясно – оператор `while`, който изпълнява абсолютно същите действия.

Задача. Фирма произвела през 2010 г. 130 тона продукция. Всяка от следващите две години производството спадало с 10%. Напишете конзолно приложение, което да определи през коя година то ще е за първи път под 10 тона?

Решение: Ще използваме две променливи – една дробна x за да помним до колко е спаднала продукцията и една цяла g – за да помним поредната година. На всяка стъпка в тялото на цикъл ще намаляваме x с 10%, а ще увеличаваме g с 1. Ще изпълняваме цикъла, **докато** съдържанието на променливата x стане по-малко от 10. На *Фиг. 2* алгоритъмът е описан с думи. Кодирайте го на езика C# и създайте от него конзолно приложение с име Firma.

```
x = 130.0
g = 2010
докато x >= 10
    x = 0.9 * x
    g = g + 1
изведи g
```

Фиг. 2.

Оператор do...while

Третият оператор за организиране на цикъл в C# е операторът `do...while`. Разликата между него и оператора `while` е, че условието за край на цикъла се проверява след всяко изпълнение на тялото на цикъла. Затова в този случай казваме, че операторът за цикъл е със *следусловие*.

Синтаксисът на оператора `do...while` е:

```
do
{ <тяло на цикъла> }
while (<условие>);
```

Задача. Напишете конзолно приложение `sq`, което да изчислява квадратен корен от зададено цяло число.

Решение: Да означим с a зададеното цяло число. Ще използваме изчислителна процедура, която доста се различава от алгоритмите, които вече познаваме. Такава процедура се нарича *числен метод*. Ще пресмятаме последователно елементите на безкрайната редица $x_1, x_2, \dots, x_i, \dots$ по следното правило: първият член на редицата е половината на a , т.е. $x_1 = a/2$, а всеки следващ член на редицата x_{i+1} се получава от предишния x_i по формулата $x_{i+1} = (x_i + a/x_i)/2$, т.е. $x_2 = (x_1 + a/x_1)/2, x_3 = (x_2 + a/x_2)/2$ и т.н. Такъв начин на задаване на елементите на безкрайна редица се нарича *рекурентна зависимост*. Доказано е, че всеки следващ елемент на такава редица е все по-близо и по-близо да квадратния корен на a .

Очевидно е, че елементите на редицата трябва да бъдат пресмятани в цикъл. Проблемът при такава изчислителна процедура е, **кога да се прекрати изпълнението на цикъла**. Тъй като стойностите на редицата непрекъснато намаляват (опитайте се да докажете това!), както и разликата между всеки два последователни елемента на редицата, цикълът може да спре, когато разликата между два съседни члена на редицата стане по-малка или равна на някакво много малко число. Това число се нарича *приближение* или *точност* на метода. С това ограничение върху броя на стъпките на цикъла получаваме алгоритъма от *Фиг. 3*.

По традиция точността на числения метод се означава с гръцката буква ϵ (епсилон) и затова ще дадем на съответната променлива името `eps`. Последователните стойности на елементите на редицата ще съхраняваме в променливата x .

Забележете ролята на променливата `xs` – в първия оператор от тялото на цикъла в нея се съхранява предишната стойност на x , а в следващия оператор в x се изчислява новата стойност. Цикълът трябва да продължи да се изпълнява докато разликата в две последователни стойности стане по-малка от `eps`.

- 1) нека `eps = 0.001`
- 2) въведи a
- 3) $x = a/2.0$
- 4) повтаряй
- 5) $xs = x$
- 6) $x = (x + a/x)/2$
- 7) докато $(xs - x) >= eps$
- 8) изведи x

Фиг. 3.

Да проследим изпълнението на алгоритъма за $a = 9$:

$eps = 0.001$

$a = 9$

$x = 9/2 = 4.50$

Първа стъпка на цикъла:

4. $xs = 4.50$

5. $x = (4.50 + 9/4.50)/2 = 3.25$

6. $4.50 - 3.25 = 1.25 > 0.001 \rightarrow$ връщаме се на ред 4.

Втора стъпка на цикъла:

4. $xs = 3.25$

5. $x = (3.25 + 9/3.25)/2 = 3.0096153846\dots$

6. $3.25 - 3.0096153846\dots = 0.24\dots > 0.0001? \rightarrow$ връщаме се на ред 4.

Трета стъпка на цикъла:

4. $xs = 3.0096153846\dots$

5. $x = (3.0096153846\dots + 9/3.0096153846\dots)/2 = 3.00001536\dots$

6. $3.0096153846\dots - 3.00001536\dots = 0.0096\dots > 0.001? \rightarrow$ връщаме се на ред 4.

Четвърта стъпка на цикъла:

4. $xs = 3.0000153846\dots$

5. $x = (3.0000153846\dots + 9/3.0000153846\dots)/2 = 3.00000000003932$

6. $3.0000153846\dots - 3.00000000003932 = 0.00001536\dots < 0.001? \rightarrow$ край

Затова

извеждаме $x = 3.00000000003932$.

Забележете как доста бързо стойността в променливата x се доближава до квадратния корен на 9. Напишете конзолно приложение, което реализира този алгоритъм. Компилирайте го и проверете работоспособността му. Като при всеки числен метод стойността на резултата е намерена **приблизително**, или с **точност** eps . Дайте по-малка стойност на eps , например 0.0001 или 0.00001 и вижте как се променя резултатът.

Логически операции

Не е невъзможно да се моделира работата на оператора за цикъл със следусловие, чрез познатите ни вече оператори `for` и `while`, макар че това ще бъде доста изкуствено, защото при тях условието се проверява преди да се изпълни тялото на цикъла. На *Фиг. 4* е показан схематично оператор `while`, който изпълнява същото, което и операторът `do...while`. Както се вижда, налага се да организираме безкраен цикъл – с условие булевата константа `true`, който да прекратим с оператора `break`, когато условието, при което продължаваме изпълнението на цикъла вече не е изпълнено. За целта използваме не булевият израз, който е в оператора `do...while`, а неговото *отрицание*. Операцията отрицание познаваме от уроците за електронни таблици. Да си припомним и другите логически операции.

Досега получавахме логически (булеви) изрази от операциите за сравняване. Булевите изрази могат да се комбинират в по-сложни булеви изрази с помощта на *операциите*: конюнкция („И“) със знак `&&`, дизюнкция („ИЛИ“) със знак `||` и отрицание („НЕ е вярно, че ...“) със знак `!`. Тези три операции са достатъчни за построяване на произволен логически израз. Как се пресмята стойността на резултата за отделните булеви операции, можем да си припомним от таблицата на *Фиг. 5*.

```
while (true)
{
  <тяло на цикъла>
  if(!<условие>) break;
}
```

Фиг. 4.

x	y	!x	!y	x y	x&&y
false	false	true	true	false	false
false	true	true	false	true	false
true	false	false	true	true	false
true	true	false	false	true	true

Фиг. 5. Логически (булеви) операции

Например, стойността на израза $(6 > 0) \wedge (3 > 0)$ е **true**, защото е конюнкция на два израза – $6 > 0$ и $3 > 0$ – стойностите на които са **true**. От друга страна, стойността на израза $(6 > 0) \vee (3 = 0)$ също е **true**, защото е дизюнкция на два израза, стойността на единият от които – $6 > 0$ – е **true**. Изразът $(x \geq a) \wedge (x \leq b)$ описва условието: числото съхранено в променливата x да е в интервала $[a; b]$. Същото условие може да се запише и с използване на отрицанието, като знаем, че отрицанието на $x \geq a$ е $x < a$, а на $x \leq b$ – $x > b$. Получаваме еквивалентния израз $\neg((x < a) \vee (x > b))$.

И в булевите изрази, скобите се употребяват за да определят явно реда на прилагане на операциите. За да се избегне употребата на много скоби, са въведени приоритети и на булевите операции. Най-висок приоритет има отрицанието, а конюнкцията и дизюнкцията са с еднакъв приоритет и се изпълняват отляво надясно. Аритметичните операции и операциите за сравняване имат по-нисък приоритет от отрицанието, но по-висок от останалите две булеви операции. Препоръчваме на начинаещия програмист, при най-малкото съмнение за съотношението на приоритетите, да направи справка в таблицата (Урок 12-13) или, което е много по-просто и надеждно – да постави двойка скоби, които еднозначно определят реда на прилагане на операциите.

Въпроси и задачи

1. Дайте примери за изречения на български език, в които се използва предусловие и следусловие. Например, „Докато вали дъжд няма да излизам навън“.
2. Напишете булев израз в C#, който съответства на следното твърдение:
 - а) числата a, b и c са положителни;
 - б) числото x е в числовия интервал $[3; 10]$;
 - в) числото p е четно или положително;
 - г) числата a, b и c са равни по между си;
 - д) числата a, b и c са различни;
 - е) поне две от числата a, b и c са положителни;
 - ж) нито едно от числата a, b и c не е отрицателно;
 - з) точно две от числата a, b и c са равни по между си;
 - и) числата x и y са координати на точка от първи квадрант на координатната система.
3. Колко пъти ще се изпълни тялото на цикъла от *Фиг. 3*?
4. Променете програмата `sqg` така, че да показва и междинните резултати, както е в текста на урока.

36
37

Калкулатор за дроби

През следващите два часа ще започнем реализацията на по-сериозен проект – ще напишем програма с графичен интерфейс, която да извършва аритметичните действия с дроби – събиране, изваждане, умножение и деление.

Да започнем със събиране на дроби. За да изясним алгоритъма за събиране на дроби, който ще използваме, да разгледаме следния пример: На *Фиг. 1* схематично е показан един упростен алгоритъм за намиране сумата на две дроби. Нека, в общия случай, искаме да намерим сумата. Най-лесно би било, вместо да търсим най-малко общо кратно (НОК) на b и d , да вземем за общ знаменател произведението $b \cdot d$, в примера $3 \cdot 9 = 27$. Получаваме следния **алгоритъм**:

1. $q = b \cdot d$. 2. $p = a \cdot q / b + c \cdot q / d$.

За упражнение, напишете конзолно приложение, което въвежда четирите цели числа a, b, c и d , пресмята с упростения алгоритъм и извежда в конзолата числителя p и знаменателя q на сумата.

Упростеният алгоритъм не е добър за създаване на калкулатора за дроби, защото, първо, полученият резултат би трябвало да се съкрати, ако може. Както в нашия пример, където резултатът трябва

$$\begin{array}{r} \overbrace{1}^9 \quad \overbrace{2}^3 \\ \frac{1}{3} + \frac{2}{9} = \frac{15}{27} \\ \underbrace{\hspace{1.5cm}}_{27} \end{array}$$

Фиг. 1.

```

докато (a != b)
{ ако (a > b) a = a - b
  иначе b = b - a
}
НОД е a

```

Фиг. 2.

да се съкрати на 3: . По-важното е обаче, че при намиране на общ знаменател по този начин, може да се получат големи числа. Например, ако $b = 20000$, а $d = 10000$, тогава общият знаменател ще е $q = 200000000$, докато $\text{НОК}(20000, 10000) = 20000$.

В нашия калкулатор за дроби ще реализираме класическия алгоритъм, с НОК като общ знаменател. Алгоритъмът за намиране на НОК, който се изучава в училище, може да се реализира програмно, но е труден за програмиране и бавен. Затова ще използваме *алгоритъма на Евклид* за намиране на най-големия общ делител на две числа (НОД) и, че $\text{НОК}(a, b) = a \cdot b / \text{НОД}(a, b)$.

Алгоритъмът на Евклид за намиране НОД на две числа a и b е показан на Фиг. 2.

Например, ако $a = 12$ и $b = 16$, тогава алгоритъмът ще пресметне последователно следните стойности за a и b : **12** и **4** = 16 - 12; **8** = 12 - 4 и **4**; **4** = 8 - 4 и **4**. Следователно $\text{НОД}(12, 16) = 4$ и в този случай $\text{НОК}(12, 16) = 12/4 \cdot 16 = 48$.

Работа с компютър

Въведете функция `int NOD(int a, int b)`, която има два аргумента (параметъра) и връща като резултат НОД на зададените като аргументи стойности (Фиг 3а). Въведете функцията `int NOK(int a, int b)`, която също има два аргумента и връща като резултат НОК на зададените като аргументи стойности (Фиг 3б). Напишете конзолно приложение, във функцията `Main` на което да се въвеждат стойностите на a и b , да се намира техния НОК с извикване на метода `NOK` и да се извежда конзолата.

```

int NOD(int a, int b)
{ while (a != b)
  { if (a > b) a = a - b;
    else b = b - a;
  }
  return a;
}

```

a.

```

int NOK(int a, int b)
{ int c = NOD(a, b), d;
  if (a >= b) d = a / c * b;
  else d = b / c * a;
  return d;
}

```

б.

Фиг. 3.

```

въведете a, b, c и d
съкратете a/b и c/d
q = NOK(b, d)
p = a*(q/b) + c*(q/d)
съкратете p/q
изведете p и q

```

Фиг. 4.

Забележете как при намирането на НОК първо делим едно от числата на НОД и умножаваме резултата по второто. Това е за да на стане произведението на двете числа много голямо и да не може да се събере в променливата от избрания тип. По същите причини може да направим така, че да делим не кое да е, а по-голямото от двете числа на НОД.

Сега вече можем да напишем алгоритъма за намиране на сумата (Фиг. 4). Ясно е, че $q = \text{НОК}(b, d)$, а $p = a \cdot (q/b) + c \cdot (q/d)$. За да можем да осигурим пресмятанията в повече случаи, добре е да

съкратим двете дадени дроби след въвеждането им. След намиране на p и q , трябва да се провери дали и те не се съкращават. Напомяме, че за да направим една дроб несъкратима, трябва да разделим и числителя и знаменателя на техния НОД.

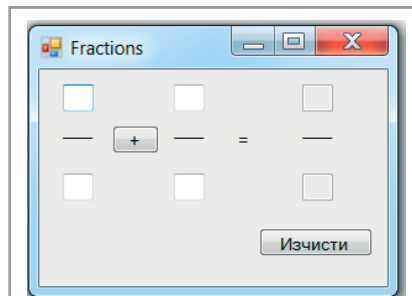
Забеляхте ли колко пъти се използва намиране на НОД в алгоритъма? Три пъти! Затова имаше смисъл намирането на НОД да се оформи като отделна функция. Забележете, че съкращаването на дроб също се използва 3 пъти, което означава, че също може да се изнесе в отделна функция.

Работа с компютър

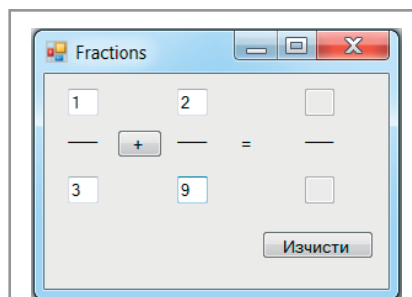
Сега вече можем да се заемем със създаване на приложението. Отворете в средата ново приложение с графичен интерфейс (Windows Forms Application) с име `Fraction`. Във формата поставете шест текстови кутии – `textBox1` – за a , `textBox2` – за b , `textBox3` – за c , `textBox4` – за d , `textBox5` – за p и `textBox6` – за

q. Поставете и два бутона – button1 за стартиране на пресмятането, надписан със знака + и button2 за почистване на текстовите кутии, преди задаване на нов пример, надписан с Изчисти. Формата трябва да е подобна на това, което е показано на *Фиг. 5*. Текстовите кутии за отговора трябва да са недостъпни за писане (`Enabled=false`). Ето как трябва да изглежда примерен програмен код на приложението:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        int NOD(int a, int b)
        {
            while (a != b)
            {
                if (a > b) a = a - b;
                else b = b - a;
            }
            return a;
        }
        int NOK(int a, int b)
        {
            return a / NOD(a, b) * b;
        }
        private void button1_Click(object sender, EventArgs e)
        {
            int a = int.Parse(textBox1.Text);
            int b = int.Parse(textBox2.Text);
            int c = int.Parse(textBox3.Text);
            int d = int.Parse(textBox4.Text);
            int n = NOD(a,b); a = a / n; b = b / n;
            n = NOD(c,d); c = c / n; d = d / n;
            int q = NOK(b, d);
            int p = a * (q / b) + c * (q / d);
            n = NOD(p, q); p = p / n; q = q / n;
            textBox5.Text = p.ToString();
            textBox6.Text = q.ToString();
        }
        private void button2_Click(object sender, EventArgs e)
        {
            textBox1.Clear(); textBox2.Clear(); textBox3.Clear();
            textBox4.Clear(); textBox5.Clear(); textBox6.Clear();
        }
    }
}
```



Фиг. 5.



Фиг. 6.

Въведете програмата, компилирайте я и проверете работоспособността ѝ (*Фиг. 6*).

Въпроси и задачи

1. На *Фиг. 7* е показана друга версия на алгоритъма на Евклид за намиране на НОД. Заменете в програмата `Fractions` старата версия на алгоритъма на Евклид с новата. Компилирайте програмата и проверете работоспособността ѝ. Кой от двата алгоритъма ще работи по-бързо и защо?
2. Довършете програмата `Fractions`. Добавете бутона за стартиране на изваждането, умножението и делението на дроби и напишете съответни методи за събитието `Click` на тези бутона. Проверете работоспособността на всички нови функции на програмата.
3. Напишете конзолно приложение, което да изведе в конзолата таблица за превръщане на температура, зададена в градуси по Фаренхайт – в градуси по Целзий. Температурите по Фаренхайт да са в интервала от 0°F до 200°F със стъпка 5.

```
ако (a < b) разменете ги
докато (a % b != 0)
{
    r = a % b
    a = b
    b = r
}
код е b
```

Фиг. 7.

Упътване. Формулата за преобразуване от Фаренхайт в Целзий изведете от формулата за преобразуване от Целзий във Фаренхайт, която използвахме в предишен урок.

4. ■ Реколтата от ябълки през 1990 година е била 20 тона. След това, на всеки две години реколтата намалява с 20%. Направете конзолно приложение, което решава следните задачи:
- намира годината, през която за първи път реколтата е била по-малка от 5 тона;
 - намира годината, през която сумарното количество ябълки ще превиши 90 тона.

38

Егномерен масив

Да се опитаме да решим с програма следната, не лека за изпълнение на ръка, задача: Дадени са средните дневни температури, измерени в град София, за 2011 г. Да се намерят десетте най-високи стойности. Вече сме решавали задача, при която се въвеждат определен брой числа и се намира най-голямото от тях, чрез преглеждане на всички числа. В случая обаче се иска да изведем не най-голямото число, а десетте най-големи, което налага да се прегледат дадените 365 числа 10 пъти. **Не можем да искаме от потребителя да въвежда данни за програмата повече от веднъж! Ако някои от въведените данни ни трябва повече от един път, те трябва да се съхраняват в паметта.** Абсолютно неразумно би било също така, да декларираме 365 променливи – a_1, a_2, \dots, a_{365} и да пишем отделен код за обработката на всяка от тях. Тъй като описаната ситуация е много характерна, всеки език предоставя средства за справяне с проблема. С тези средства ще се запознаем в този урок.

Масиви

За решаването на тази, и подобни на нея задачи, в езиците за програмиране се използват *масиви*. Масивът е област от паметта (*структура*), разделена на *полета* с еднакъв размер, в които могат да се съхранят едновременно **редица от стойности от един и същ тип**. Масивът има *име*, което е общо за всички полета на структурата и *дължина* – броят на полетата. Всяко поле от съответния на масива тип, наричано *елемент* на масива, може да се разглежда и използва в програмата **като променлива от същия тип**. Елементите на масива се различават един от друг по своя *индекс* – така наричаме поредния номер на елемента в редицата. Номерирането на елементите на масив в C# започва от нула. На *Фиг. 1* е показан масив a с 8 елемента от тип `int`.

В езика C# името на елемент на масив се образува, като след името на масива се постави индексът му, ограден в квадратни (средни) скоби: $a[0], a[1], \dots, a[i], \dots$

a	1	-3	5	5	2	-6	4	11
	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$

Фиг. 1.

Деклариране на масив

Както променливите, така и масивите трябва да се декларират.

Синтаксисът на оператора за **деклариране на масив** е

`<тип>[] <име на масива>;`

Например, `int[] a; double[] point; string[] names;`. В един оператор могат да се декларират няколко масива. Например, `int[] a, b, c;`

За разлика от декларирането на променливи, **декларацията на масив не означава автоматично заделяне на памет за масива** по време на изпълнение на програмата, както е с обикновените (*скаларните*) променливи. Това се подсказва и от факта, че никъде в оператора за деклариране на масив не се указва размера му. Затова, преди да се използва един масив той трябва да се разположи в паметта с помощта на оператора за заделяне на памет.

Синтаксисът на оператора за **заделяне на памет** за деклариран масив е

```
<име на масива> = new <тип>[<размер>];
```

Например, `a = new int[8]; point = new double[2];`. Подобно на инициализацията на скаларните променливи, заделянето на памет може да стане в оператора за деклариране на масива. Например, `int[] a = new int[8];`.

След като е указана дължината на масива, например *n*, в програмата могат да се използват само елементи с индекси от 0 до *n* - 1!

Инициализация на масив и извеждане на елементите

Инициализацията на масив се състои в инициализация на елементите му. По подразбиране, елементите на целочислените масиви се инициализират с нули, масивите от тип `char` – с малки латински букви 'a', а от тип `string` – с празни низове. Ако потребителят иска друга инициализация, има няколко начина да се направи това.

❖ **В оператора за деклариране.** Например, `int[] a = {1, -3, 5, 5, 2, -6, 4, 11};`. При този начин, стойностите на елементите се разделят един от друг със запетая и се заграждат в къдрави скоби. В този случай няма нужда от служебната дума `new` и задаване на размера. За масива се отделя толкова полета в паметта, колкото са елементите в инициализацията списък.

❖ **В кода на програмата.** В този случай даването на стойност става с оператор за присвояване за всеки елемент поотделно и не се различава от даването на стойност на скаларна променлива. Например, `a[0] = 1; a[1] = -3; a[2] = 5;` и т.н. или в цикъл `for(i=0;i<n;i++) a[i] = i;`.

❖ **Въвеждат се от потребителя,** по време на изпълнение на програмата. За образец може да се използва следният програмен фрагмент:

```
int n, i;
n = int.Parse(Console.ReadLine());
int[] array = new int[n];
for (int i = 0; i < n; i++)
{ array[i] = int.Parse(Console.ReadLine()); }
```

Забележете, възможността, която се предоставя на програмиста в този случай – да остави на потребителя сам да определи размера на масива по време на изпълнението. В примера, броят на елементите на масива се въвежда в променливата *n* и след това се декларира самият масив, с размер *n*.

Извеждането на елементите на масив също не се отличава от извеждането на коя да е скаларна променлива от същия тип. За образец може да се използва следният програмен фрагмент:

```
for (int i = 0; i < 5; i++)
{ Console.WriteLine(a[i]); }
```

Работа с компютър

За илюстрация на използването на масив да решим следната **задача**. Зададени са 10 цели числа в определен ред. Да се въведат в компютъра и да се изведат на конзолата в ред, обратен на реда на въвеждането им.

Решение: Очевидно е, че числата трябва да се съхранят в масив. Задачата може да се реши по 2 начина. Първата възможност е да се въведат числата в масива с цикъл, в който управляващата променлива се мени от 0 до 9, а при извеждането – да намалява от 9 до 0. Вторият начин е да се постъпи обратно: при въвеждането управляващата променлива на цикъла да се мени от 9 до 0, а при извеждан – от 0 до 9. Програмният фрагмент с който реализираме първият подход е:

```

int i, n=10;
int[] array = new int[n];
for (i = 0; i < n; i++)
{ array[i] = int.Parse(Console.ReadLine());
}
for (i = n-1; i >= 0; i--)
{ Console.WriteLine(array[i]);
}

```

Напишете съответно конзолно приложение, компилирайте го и проверете работоспособността му.

Въпроси и задачи

- За всеки от програмните фрагменти:
 - `int[] a = {1,2,3,4,5,6}, b;` б. `int[] a = {1,2,3,4,5,6}, b;`
`b = a[2] + a[3];` `b = a[2] + a[6];`
 определете стойността на **b** след изпълнението на фрагмента?
- Дадено са *n* цели числа. Напишете конзолно приложение, което въвежда от клавиатурата броя на числата и самите числа и извежда в конзолата само отрицателните числа, в ред обратен на реда на въвеждането им.

ПРИМЕР:

Вход	Изход
3	-2
-4	-4
1	
-2	

- Дадени са две редици от по *n* числа. Напишете конзолно приложение, което въвежда броя *n* и елементите на редиците, а извежда на конзолата YES, ако двете редици са съставени от едни и същи числа, в един и същ ред, или NO в противен случай.

ПРИМЕР 1:	Вход	Изход	ПРИМЕР 2:	Вход	Изход
	3	YES		3	NO
	1			1	
	1			1	
	-2			-2	
	1			1	
	1			-2	
	-2			1	

Масивите са много важни за програмирането. За много задачи данните трябва да се разполагат в масив, за да могат да се обработват ефективно. Вече знаем как се декларира едномерен масив, как се въвеждат и извеждат елементите му. В този и следващи уроци ще покажем редица техники за работа с масиви, които са необходими за решаване на по-сложни задачи, като например:

- ❖ да се намери най-големия и най-малкия елемент (ако елементите са от тип, който позволява сравняване по големина);
- ❖ да се търси елемент с конкретна стойност;
- ❖ да се добавя елемент;
- ❖ да се премахва елемент/елементи по даден признак;
- ❖ да се подредят (сортират) елементите по даден признак, и т.н.

Задача 1: Дадена е редица от n цели числа. Напишете конзолно приложение което да намира и извежда в конзолата най-голямото и най-малкото число на редицата.

Решение: Вече знаем как се намира най-голямото число на редица, която въвеждаме от клавиатурата, без да използваме масив. Най-малкото число намираме по същия начин – приемаме първото число за текущо най-малко, а на всяка стъпка на цикъла заменяме текущо най-малкото с нововъведеното, ако то се окаже по-малко от текущото. Причината да поискаме числата да се съхранят в масив е, че за някои по-сложни задачи може да се наложи числата да се прегледат отново, а не е добре да искаме от потребителя на програмата да въвежда числата втори път.

Програма:

```
static void Main(string[] args)
{
    int i, n, maxN, minN;
    n = int.Parse(Console.ReadLine());
    int[] array = new int[n];
    array[0] = int.Parse(Console.ReadLine());
    maxN = array[0]; minN = array[0];
    for (i = 1; i < n; i++)
    {
        array[i] = int.Parse(Console.ReadLine());
        if (maxN < array[i]) maxN = array[i];
        if (minN > array[i]) minN = array[i];
    }
    Console.WriteLine("Max =" + maxN);
    Console.WriteLine("Min =" + minN);
}
```

Създайте конзолно приложение MaxMin. Въведете програмата, компилирайте я и проверете работоспособността ѝ.

Тестване: Когато тествате програма, в които трябва да се въвеждат дълги редици от числа, никога не задавайте голяма стойност за n . Защо? Да допуснем, че в програмата има грешка. Стартирайте я с $n = 10$ и въвеждате 10 числа. Забелязвате грешката, поправяте я и отново стартирате програмата. Пак въвеждате 10 числа. А ако в програмата все още има грешки ...? Много по-добре е да тествате с малко n , например 4 числа вместо 10, докато се убедите че сте изчистили всички грешки? Много е вероятно програмата, която работи правилно за $n = 4$, да работи правилно и за $n = 10$, $n = 100$, $n = 1000$ и т.н. Това не значи, че не трябва да направите и 1-2 по-големи теста.

Задача 2: Дадена е редица от n цели числа. Напишете конзолно приложение което да намира и извежда в конзолата най-малкото и най-голямото число на редицата и по колко пъти се срещат в нея тези две числа.

Решение: За решаването на тази задача трябва само да се допълни кодът от решението на **Задача 1**. Необходимо е още едно обхождане на елементите на масива с цикъл, в който ще се преброява колко пъти се срещат запомненият в променлива `maxN` максимален елемент и запомненият в променлива `minN` минимален елемент на масива. За целта са необходими две нови променливи `brMin` и `brMax` – броячи, които ще увеличаваме с 1 всеки път когато срещнем най-малкото или най-голямото число, съответно. Както винаги, когато декларираме променлива брояч, не трябва да забравяме да я инициализираме с нула.

Създайте ново конзолно приложение `brMinMax` като копирайте в него кода от **Задача 1** и добавите в края на функцията `Main` следните допълнителни редове:

```
int brMin = 0, brMax = 0;
for (i = 0; i < n; i++)
{
    if (minN == array[i]) brMin++;
    if (maxN == array[i]) brMax++;
}
Console.WriteLine("brMin=" + brMin);
Console.WriteLine("brMax=" + brMax);
```

Компилирайте програмата и проверете работоспособността ѝ.

Задача 3. Съставете алгоритъм за решаване на **Задача 2**, който не използва масив. Напишете конзолно приложение, което реализира този алгоритъм.





Решение: Да се спрем само на намиране на броя на срещанията на най-малкото число, защото за най-голямото ще постъпим по подобен начин. На пръв поглед изглежда невъзможно да се реши задачата без използване на масив, защото ще знаем кое е най-малкото число, едва когато завършим с въвеждането на всички числа. Но не е така! Да се възползваме от идеята, която при решаването на **Задача 1** нарекохме **текущо минимално**. След като в променливата `minN` запомняме най-малкото намерено до момента число на редицата, тогава нека паралелно с това в променливата `brMin` да съхраняваме броя на срещанията на това текущо най-малко число. Когато на поредната стъпка на цикъла срещнем текущото минимално, тогава трябва да увеличим съдържанието на `brMin` с 1. А когато срещнем число, което е по-малко от текущо минималното? Тогава просто трябва да започнем броенето отново, като дадем на `brMin` стойност 1.

Програма:

```
static void Main(string[] args)
{
    int i, n, a, minN, brMin;
    n = int.Parse(Console.ReadLine());
    minN = int.Parse(Console.ReadLine());
    brMin=1;
    for (i = 1; i < n; i++)
    {
        a = int.Parse(Console.ReadLine());
        if (minN > a) { minN = a; brMin = 1; }
        else if (minN == a) brMin++;
    }
    Console.WriteLine("Min =" + minN);
    Console.WriteLine("brMin=" + brMin);
}
```

Компилирайте програмата и проверете работоспособността ѝ.

Въпроси и задачи

1.  Дадена е редица от n цели числа. Напишете конзолно приложение което въвежда числата и извежда в конзолата най-голямото число на редицата и на коя позиция (бройки от 0) то се среща за първи път.
2.  Добавете към решението на предната задача код, който да извежда и най-малкото число на редицата и позицията, на която то се среща за първи път.
3.  Променете решението на **Задача 3** от урока така, че да намира и броя на срещанията на най-голямото число в редицата.
4.  Напишете конзолно приложение, в което се въвеждат оценките от контролна работа по информатика на 13-те ученици от една група и се намира броят на учениците, които имат оценки по-големи от средният успех на групата (представи ли са се добре!).

Задачата за сортиране познаваме от уроците за електронни таблици. Няма съмнение, че задачата за сортиране е една от най-важните в Информатиката. Решенията на редица други задачи стават много по-прости и бързи, ако данните са предварително сортирани. Разработени са много алгоритми за сортиране с различни качества. В този урок ще запознаем с един от тях. За целта трябва първо да изясним програмната конструкция наречена *вложени цикли*.

На *Фиг. 1* е показана част от правоъгълна координатна система в равнината и правоъгълник с върхове в точките с координати (1,1), (4,1), (4,3) и (1,3). От уроците по ИТ знаем, че екранът на компютъра е правоъгълник разделен на малки квадратчета, наречени *пиксели*. Затова можем да си мислим, че на фигурата е изобразена координатната система на екрана и е избран правоъгълник съставен от 6 пиксела. За координати на пиксел ще считаме координатите на долния му ляв ъгъл.

За решаването на задача, свързана с изображения на екрана правоъгълник се налага да посетим в някакъв ред пикселите, съставлящи правоъгълника. Това може да направим, като „обходим“ последователно стълбовете на правоъгълника, започвайки от този с x -координата 1, а във всеки стълб „обходим“ последователно пикселите на стълба също започвайки от този с y -координата 1. Ако трябва да направим програмно това, за обхождането на стълбовете ще ни трябва цикъл, в който променливата x , в която ще помним координатата на стълба, се мени от 1 до 3. Естествено това да е цикъл `for`:

```
for (x = 1; x <= 3; x++) { ... }
```

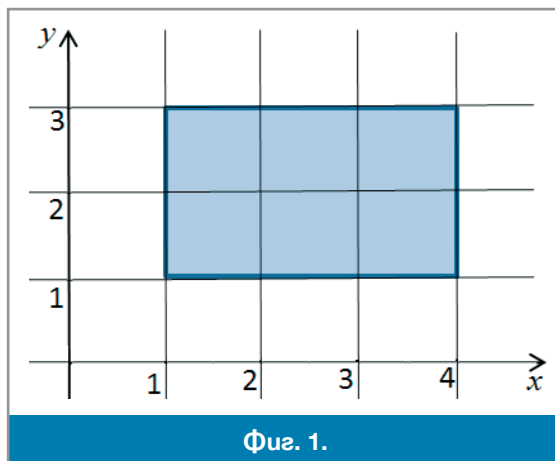
За обхождането на пикселите в стълба с координата запомнена в променливата x също трябва да направим цикъл, в който в който променливата y , в която ще помним координатата на реда да се мени от 1 до 2. Това също ще бъде цикъл `for`:

```
for (y = 1; y <= 2; y++) { ... },
```

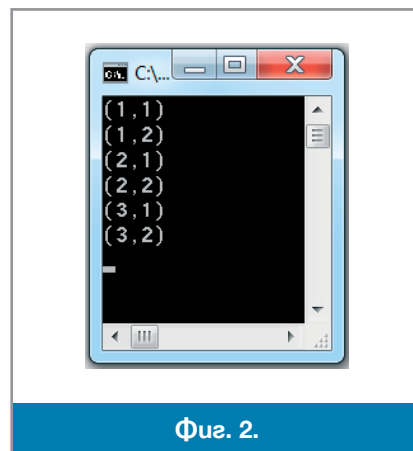
който трябва да бъде част от тялото на първия `for`. Казваме, че вторият цикъл е *вложен в първия*. В тялото на втория цикъл изписваме действията, които трябва да се извършат с пиксела, чиито координати са в x и y . За примера е достатъчно да изведем в конзолата координатите на пиксела. Получаваме програмния фрагмент:

```
for (int x = 1; x <= 3; x++)
    for (int y = 1; y <= 2; y++)
        Console.WriteLine("{0},{1}", x, y);
```

Цикълът по x се нарича *външен*, а този по y – *вътрешен*. Ако вътрешният цикъл е единствен оператор на тялото на външния, тогава фигурни скоби за ограждане на вътрешния цикъл не са необходими. Резултатът от изпълнението на този фрагмент е показан на *Фиг. 2*.



Фиг. 1.



Фиг. 2.

Алгоритъм на мехурчето

Ще разгледаме популярния алгоритъм за сортиране – *алгоритъма на „мехурчето“*. Идеята на този алгоритъм е следната. Когато в газирана течност мехурче с газ, издигайки се нагоре се сблъска с по-леки от него, то ги измества и продължава нагоре, а когато срещне по-тежко от него, тогава лекото остава на място, а тежкото тръгва нагоре и стига до края, ако не срещне друго, още по-тежко, разбира се.

Нека е даден масив a с n елемента. Започваме с „мехурчето“ намиращо се в $a[0]$. Сравняваме го с елемента в $a[1]$ и ако $a[0] > a[1]$ – разменяме им местата. Повтаряме същото за $a[1]$ и $a[2]$, за $a[2]$ и $a[3]$, и т.н. докато стигнем до $a[n-2]$ и $a[n-1]$. По този начин най-големият елемент застава на последно място. Започваме отново от началото на масива, но този път без да отиваме до края, тъй като там вече е най-големият елемент на масива. Затова последното сравнение е на елементите $a[n-3]$ и $a[n-2]$. Продължаваме по същия начин като на всеки следващ етап свършваме с един елемент по-рано. За последния етап остава да се сравнят само $a[0]$ и $a[1]$.

Да разгледаме следния пример. Даден е масив с четири елемента `int a[] = {3,2,1,4}`. В таблицата на *Фиг. 3*, по колони са проследени промените в стойностите на елементите на масива по време

на сортирането, като сравняваните два елемента са в червено. В първия ред на таблицата са извършваните сравнявания. Тези от тях, които водят до разместване, са показани също в червено.

Да разпишем подробно извършваните от алгоритъма сравнявания:

на първи етап: $a[0]$ с $a[1]$, $a[1]$ с $a[2]$ и $a[2]$ с $a[3]$ (3 сравнявания)

на втори етап: $a[0]$ с $a[1]$ и $a[1]$ с $a[2]$ (2 сравнявания)

и накрая: $a[0]$ с $a[1]$ (1 сравняване)

3>2	3>1	3<4	2>1	2<3	1<2
3	2	2	2	1	1
2	3	1	1	2	2
1	1	3	3	3	3
4	4	4	4	4	4

Фиг. 3.

Очевидно е, че трябва да използваме два вложени един в друг цикъла, за да извършим сортирането. Тъй като алгоритъмът всеки път сравнява елемент със следващия го, достатъчно е да определим как се менят индексите само на първия от сравняваните елементи (показан в червено). Забелязваме че на първия етап последният сравняван елемент е $a[n-2]$, а във всеки следващ етап, вътрешният цикъл трябва да свършва с един елемент по-рано. Затова във външния цикъл

управляващата променлива, например i , ще меним от $n-2$ до 0 , намалявайки я всеки път с 1 , а във вътрешния управляващата променлива, например j , ще меним от 0 до i , увеличавайки я всеки път с 1 :

```
int i, j, t, n;
Console.WriteLine("Въведете n и n-те числа:");
n = int.Parse(Console.ReadLine());
int[] a = new int[n];
for (i = 0; i < n; i++) // Въвеждане
    a[i] = int.Parse(Console.ReadLine());
for (i = n-2; i >= 0; i--) // Сортиране
    for (j = 0; j <= i; j++)
        if (a[j]>a[j+1])
            { t=a[j]; a[j]=a[j+1]; a[j+1]=t; }
for (i = 0; i < n; i++) // Извеждане
    Console.WriteLine(a[i]);
```

Много важна характеристика на алгоритмите е тяхното *бързодействие*. В алгоритмите съдържащи цикли, много важно за бързодействието е, колко пъти се изпълнява тялото на цикъла. Да преброим броя на изпълненията на тялото на вътрешния цикъл в дадения пример с четири елемента: $3 + 2 + 1 = 6$. За масив с n елемента ще имаме съответно: $(n-1) + (n-2) + \dots + 2 + 1 = n \cdot (n-1)/2$ изпълнения. Например, при $n = 10\,000$, тялото на вътрешния цикъл ще се изпълни около $50\,000\,000$ пъти. Затова алгоритъмът на мехурчето може да се използва само при сортиране на неголеми масиви. Съществуват по-бързи алгоритми за сортиране, но няма да ги разглеждаме тук.





Работа с компютър

Създайте конзолно приложение `Bubble` с дадения по-горе код. Компилирайте програмата и я изпълнете за да проверите работоспособността ѝ.

Въпроси и задачи

- Забележете, че ако зададете на алгоритъма на мехурчето сортиран масив, то броят на изпълненията на тялото на вътрешния цикъл ще бъде отново $n \cdot (n-1)/2$. Какво трябва да се направи за да не извършва алгоритъмът толкова много ненужни проверки? Напишете програма която работи много по-бързо в такъв специален случай.

Упътване: Помислете, по какво в края на първото изпълнение на вътрешния цикъл може да разберете, че масивът вече е сортиран.

2.  Напишете програма, която да сортира зададени n числа в низходящ ред.
3.  Заден е средният успех на всеки от n -те ученика в едно училище ($n > 3$). Напишете програма, която да намира и извжда трите най-високи успеха.
4.  Напишете програма, която да намира числото, което се среща най-често в редица от n числа. Ако в редицата има няколко такива числа, програмата трябва да изведе най-голямото от тях. **Упътване:** Задачата може да се реши със сортиране на масива в низходящ ред. Тогава еднаквите елементи ще са един след друг. Например, нека след сортирането масивът да изглежда така: 7 7 7 6 6 6 5 5 3 3 3 2 2 1. Отляво надясно се преброяват еднаквите поредни елементи и броят се сравнява с максималния намерен до момента брой еднакви (в началото този брой е нула): $\underbrace{7\ 7\ 7}_3\ \underbrace{6\ 6\ 6}_4\ \underbrace{5\ 5}_3\ \underbrace{3\ 3\ 3}_4\ \underbrace{2\ 2}_2\ \underbrace{1}_1$. Ако броят в поредната група еднакви е строго по-голям от запомнениия – заменяме запомнениия с новия по-голям брой, както при намиране на минимален елемент.
5.  Напишете програма, която да намира числото, което се среща най-често в редица от n числа, всичките в интервала от 0 до 99. Ако в редицата има няколко такива числа, програмата трябва да изведе най-голямото от тях.

Упътване: Задачата се различава съществено от предишната по това, че числата са малки. Затова може да се реши по-бързо, като се избегне сортирането. Използваме масив b със 100 елемента, като в елемента $b[i]$ броим колко пъти числото i се среща в редицата. Когато програмата прочете числото a , увеличава съответния брояч $b[a]++$. Решение на задачата е индексът на най-голямото от числата в масива b , а ако има няколко такива – най-голямото от тях:

```
int max = 0, cis = 0;
int[] b = new int[100];
for (int i = 0; i < n; i++)
    { int a = int.Parse(Console.ReadLine());
      b[a]++;
      if (max < b[a]) { max = b[a]; cis = a; }
      else if (max == b[a] && cis < a) cis = a;
    }
Console.WriteLine(cis)
```

Направете конзолно приложение от дадения по-горе фрагмент. Компилирайте го и проверете работоспособността му?

41
42

Двумерен масив

Понятие за двумерен масив

В едномерни масиви съхраняваме данни за които е естествено да се разполагат в линия или както казваме в математика – в редица. Така всеки елемент на едномерния масив се идентифицира с един индекс, който сочи мястото на елементите в редицата. В предишен урок, обаче, дадохме пример за обект, елементите на който е естествено да се разположат в част от равнината – това беше растерът от пиксели на екрана. За да могат да се съхраняват данните свързани с такива обекти в компютърната памет, езиките за програмиране предлагат структурата *двумерен масив*.

В двумерните масиви данните са разположени в *редове* и *стълбове*. Затова всеки елемент на масива се идентифицира с два индекса – първият е номер на реда, а вторият – номер на стълба, в който се намира елементът. В езика *C#* номерацията на редовете и стълбовете, както и номерацията в едномерните масиви, започва от нула. На *Фиг. 1а* е показан двумерен масив с три реда и четири стълба а на *Фиг. 1б* индексиранието на клетките така, както се изписва в математиката, където съответните структури се наричат *матрици*.

	0	1	2	3
0	2	1	3	11
1	5	6	7	8
2	4	12	9	10

а.

	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}

б.

Фиг. 1.

Деклариране, разполагане и инициализация

Операторът за **деклариране и заделяне памет на двумерен масив** е подобен на оператора за деклариране и заделяне на памет на едномерен:

```
<тип> [ , ] <име> = new <тип> [ <брой редове> , <брой стълбове> ] ;
```

например, `int [,] intMatrix = new int [4 , 6] ;`.

И в този случай разполагането на масива в паметта може да се отдели от декларирането, но не бива да се забравя, че **преди да бъде разположен в паметта масивът не може да се използва**. Това, по което се различават двата оператора е, че в средните скоби след задаване на типа на масива, се поставя запетая, която подсказва че ще има два индекса и значи масивът е двумерен.

И тук е възможна декларация, при която се разполага масивът и се дават стойности на елементите му. Всеки от редовете на матрицата се инициализира като едномерен масив – списък от стойностите разделени със запетая и поставени в средни скоби. Списъците от стойности на отделните редове също се разделят със запетая и се поставят в средни скоби. Например:

```
int [ , ] matrix = { { 0 , 2 , 4 , 0 , 9 , 5 } , { 7 , 1 , 3 , 3 , 2 , 1 } ,
{ 1 , 3 , 9 , 8 , 5 , 6 } , { 4 , 6 , 7 , 9 , 1 , 0 } } ;
```

Обръщение към елементите на двумерен масив

При обръщение към елемент на двумерен масив, двата индекса се поставят в средни скоби и са разделни със запетая. Например, операторът `m [2 , 3] = 12 ;` ще присвои стойност 12 на елемента на масива `m`, който се намира във втори ред и трети стълб (напомняме че номерата на редовете и стълбовете започват от 0). Въвеждането на стойностите на масив `a` с `n` реда и `m` стълба от конзолата става с **два вложени цикъла**:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        a [ i , j ] = int.Parse(Console.ReadLine());
```

Елементите на двумерен масив могат да се извеждат по различен начин. Например, в редица по редове или в редица по стълбове. Най-естествено е обаче матрицата да се изведе в конзолата, като всички елементи от един ред се извеждат на един ред на конзолата, а различните редове на матрицата на различни редове на конзолата:

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
        Console.Write(a [ i , j ] + " ");
    Console.WriteLine();
}
```

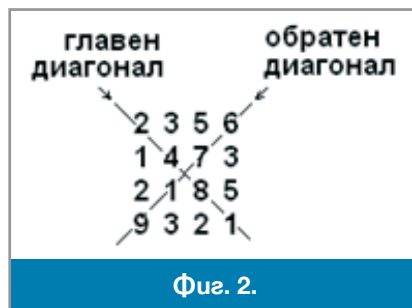
Работа с компютър

Задача 1: Напишете конзолно приложение, което въвежда елементите на матрица с размери `n` на `n` в двумерен масив и отпечатва сумите на елементите от главния и обратния диагонал (виж Фиг. 2).

Пояснение: Главен и обратен диагонал може да има само в масив с еднакъв брой редове и стълбове. В примера от *Фиг. 2* n е 4, сумата на елементите на главния диагонал е $2 + 4 + 8 + 1 = 15$, а на обратния е $6 + 7 + 1 + 9 = 23$.

Решение:

От дефиницията на главен и обратен диагонал се вижда, че елементите от главния диагонал имат еднакви индекси. За елементите от обратния диагонал сумата на индексите е постоянно число (в примера е 3). Тъй като размерността на масива в примера е $n = 4$, то очевидно в общия случай сумата на двата индекса за елементите от обратния диагонал е $n-1$. Задачата се решава само с един цикъл `for`, в който се намират едновременно и двете суми. Управляващата променлива на цикъла – брояч i – се мени от 0 до $n-1$. Така по главния диагонал сумираме елементите $a[i, i]$, а по обратния – елементите $a[i, n-1-i]$.



Напишете конзолно приложение *Diagonals*, което решава задачата, компилирайте програмата и я изпълнете за да проверите работоспособността ѝ.

Задача 2. Напишете конзолно приложение, което въвежда елементите на матрица с размери n на n в двумерен масив и отпечатва сумите на елементите под главния и над главния диагонал.

ПРИМЕР. За матрицата от *Фиг. 2* програмата трябва да изведе 18 за сумата на елементите под главния диагонал и 29 за сумата на елементите над него.

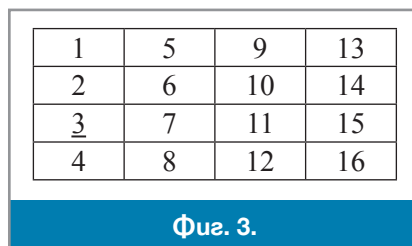
Упътване: Забележете зависимостта, която съществува между индексите на елементите под главния диагонал – първият им индекс винаги е по-голям от втория. Открийте сами зависимостта между индексите на елементите над главния диагонал. За решаването и на тази задача са нужни два вложени цикъла.

Задача 3. Напишете конзолно приложение, което въвежда елементите на матрица с размери n на n в двумерен масив и отпечатва сумите на елементите под обратния и над обратния диагонал.

ПРИМЕР. За матрицата от *Фиг. 2* програмата трябва да изведе 17 за сумата на елементите над обратния диагонал и 22 за сумата на елементите под него.

Задача 4. Напишете конзолно приложение, което генерира в двумерен масив и после извежда на конзолата матрицата показана на *Фиг. 3*.

Задача 5. Напишете конзолно приложение, което въвежда в двумерен масив оценките на n ученика по m предмета и, след като извърши необходимите пресмятания, извежда на конзолата номера на ученика с най-висок среден успех и номера на предмета с най-нисък среден успех.



ПРИМЕР: В таблицата на *Фиг. 4* са дадени примерни данни за 5 ученика и 3 предмета (клетките в бяло). За примера, програмата трябва да изведе 4 и 2.

Номер	0	1	2	Среден успех на учениците
0	3	4	3	3,33
1	5	4	3	4,00
2	3	2	3	3,00
3	5	5	6	5,33
4	6	6	5	5,67
Среден успех по предмети:	4,40	4,20	4,00	

Фиг. 4.

Въпроси и задачи

1. Напишете програма, която въвежда в двумерен масив матрица с размер n на m и извежда броя на четните и нечетните числа в матрицата.

2. Напишете програма, която въвежда в двумерен масив матрица с размер n на m и извежда сумата на елементите му.
3. Напишете програма, която въвежда в двумерен масив матрица с размер n на m и извежда сумата на елементите на ред r и стълб s (r и s също се въвеждат от клавиатурата).
4. Напишете програма, която въвежда в двумерен масив матрица с размер n на m и извежда най-големия елемент, както и номерата на реда и стълба в които се намира.
5. Напишете програма, която въвежда в двумерен масив матрица с размер n на m и извежда най-малкия елемент и колко пъти се среща в матрицата.

43 Масиви. Тест

1. Даден е масив: `int[] a = {3,1,2,5,6}`. Стойността на `a[3]` е:
а) 2; б) 5; в) 1; г) 6.
2. В следната декларация: `int[12] a = new int[6]` масивът а:
а) има 6 елемента; б) има 7 елемента;
в) има 12 елемента; в) е деклараран неправилно.
3. За програмния фрагмент `int n = 12; int[] array = new int[n];` определете най-големия индекс на елемент на масива `array`:
а) 11; б) 3; в) 12; г) фрагментът е грешен.

<pre>int[] arr = new int[5]; for(int i = 0; i < 5; i++) { arr[i] = i+3; }</pre> <p style="text-align: center;">а.</p>	<pre>int[] arr = new int[5]; int s = 0; for(int i = 0; i < 5; i++) { arr[i] = i; s = s + arr[i]; }</pre> <p style="text-align: center;">б.</p>	<pre>int[] arr = new int[5]; int s=0; for(int i = 0; i < 5; i++) { if (arr[i]>0) s = s + 1; }</pre> <p style="text-align: center;">в.</p>
--	---	---

Фиг. 1.

4. След изпълнение на програмния фрагмент от Фиг. 1а стойността на `arr[4]` е:
а) 5; б) 6; в) 7; г) друга.
 5. След изпълнение на програмния фрагмент от Фиг. 1б стойността на `s` е:
а) 5; б) 0; в) 15; г) друга.
 6. В променливата `s` програмният фрагмент от Фиг. 1в намира:
а) сумата на числата от масива; б) сумата на отрицателните числа;
в) разликата на отрицателните числа; г) сумата на положителните числа.
- В задачи 7, 8, 9 и 10 се използва декларацията от Фиг. 2а.

<pre>int[,] a = { {0,2,4,0,9,5}, {7,1,5,3,2,1}, {1,3,9,8,5,6}, {4,6,7,9,1,0} };</pre> <p style="text-align: center;">а.</p>	<pre>int m = a[0,0]; for (int i = 0; i <= 3; i++) for (int j = 0; j <= 5; j++) { if (m<a[i,j]) m=a[i,j]; }</pre> <p style="text-align: center;">б.</p>	<pre>int s=0; for (int i = 0; i <= 3; i++) { s=s+a[i,i]; }</pre> <p style="text-align: center;">в.</p>
---	---	---

Фиг. 2.

7. Елементът $a[3,2]$ има стойност:
 а) 7; б) 3; в) 5; г) друга.
8. След изпълнение на фрагмента от *Фиг. 2б* променливата m ще има стойност:
 а) 0; б) 9; в) 7; г) 5.
9. След изпълнение на фрагмента от *Фиг. 2в* променливата s ще има стойност:
 а) 1; б) 0; в) 19; г) 10.

<pre>int i,s=0; for(i = 0; i <= 3; i++) s=s+a[i,1];</pre>	<pre>int[] a = { 2, 1, 4, 3}; int i,s=1; for(i = 0; i < 4; i++) s = s*a[i];</pre>	<pre>int[] a = { 1, 5, 4, 6}; int[] b = { 2, 3, 5, 6}; int[] c = new int[4]; for (int i = 0; i < 4; i++) c[i] = a[i] + b[i];</pre>
--	--	---

а.

б.

в.

Фиг. 3.

10. След изпълнение на фрагмента от *Фиг. 3а* променливата s ще има стойност:
 а) 12; б) 1; в) 16; г) 0.
11. След изпълнение на фрагмента от *Фиг. 3б* променливата s ще има стойност:
 а) 24; б) 4; в) 1; г) друга.
12. След изпълнение на фрагмента от *Фиг. 3в* променливата $c[2]$ ще има стойност:
 а) 2; б) 8; в) 9; г) 12.

44
46

Проект: Статистика за паралелка

След като вече знаем толкова много за програмите, циклите, масивите и графичния интерфейс, крайно време е да се заемем със създаването на програмен продукт, който да е в състояние да върши нещо сериозно и полезно? Защото, да не забравяме, програмирането не е нито самоцел, нито средство да се измъчват учениците, а изключително полезно и практично занимание! Най-добрият начин да проверим доколко добре сме усвоили преподаваното е, да разработим и реализираме **проект** със задача създаване на полезна приложна програма. Още повече, че от уроците по ИТ знаем какви са основните стъпки при разработването на проект и сме реализирали проекти.

Задание

Представете си следната ситуация: Вашият класен ръководител, виждайки, че се занимавате усърдно с програмиране, ви моли да направите програма, с която да се въвежда средния успех на всеки ученик от класа ви, да се пресмята и извежда средния успех на паралелката. Това е лесна задача, с която се справяте бързо. След като сте я решили, обаче, класният ви пита: „А ще може ли да се преправи програмата така, че да извежда най-високия и най-ниския успех в класа?“. Разбира се, това също може да го направите. След като вижда колко „бързо и лесно“ се справяте със задачите, много е вероятно класният да поиска програмата да може да извежда и други справки: колко ученика имат слаб успех, колко среден, и т.н., което също не представлява проблем за вас ...

След това, добил смелост от успехите ви в програмирането, обнадежден от резултатите от вашата работа, класният предлага в програмата да се въвежда не средния успех на всеки ученик, а оценките му по всеки отделен предмет. Така програмата ще може да пресметне и средния успех на паралелката по всеки отделен предмет. Нещо повече – ще може да се пресметне и броя на оценките „слаб“, „среден“, „добър“, „много добър“ и „отличен“ по всеки предмет, както и за цялата паралелка. И какво ли още не! А всичко това е много важна и полезна информация за човек който се грижи за вашите успехи в училище. И не трябва да се учудвате, че класният ръководител я иска.

Какво стана? Този урок май изобщо не прилича на предишните? Така е. Но и на най-сериозните програмисти понякога се налага да напуснат света на математическите формули и прецизните оператори на езика за програмиране, за да могат в преговори с крайния потребител да оформят това, което в производството на софтуер се нарича *задание*. Започна се от една уж лесна програма за намиране на среден успех и се стигна до специфицирането на сериозен програмен продукт!

Представете си сега, че директорката на училище види това което сте направили и остане възхитена! Какво ще последва? Ами то е ясно – след като сте направили всичко необходимо за статистиката на една паралелка, няма да е проблем да се направи за всички паралелки от един випуск (нали? погледнато отстрани, то е едно и също ...). А после – остава само да се изведат обобщени данни по паралелки, по випуски и за цялото училище. След като е необходимо, ще трябва да се направи, нали? Колкото и да е трудно ...

План

И така, вече имаме задание за бъдещия продукт. Но забележете, че в заданието не всичко е казано ясно и точно. На някои места желанията на потребителя са непълно формулирани, понякога не е ясно дали идеите са осъществими. Трябва внимателно и в подробности **да се планира разработката**. Когато се създава сериозна програма, трябва да се обмислят всички детайли – какви входни данни са необходими, как ще се въвеждат – от конзолата или с графичен интерфейс, ако интерфейсът е графичен, какви форми ще се създават и какви компоненти трябва да има във всяка форма, какво ще се извежда, какви алгоритми ще се използват за различните обработки, и т.н.

Да започнем с вида на приложението, което ще създадем. Програмата трябва да бъде приложение, с графичен интерфейс. Не само поради неоспоримия факт, че ще е по-красива, но и защото така ще бъде по-удобна за потребителя. Въвеждане на много данни предполага допускане на грешки при въвеждане и необходимост от **въвеждане на корекции**. Ако въвеждането на данни става от конзолата, не може да се използват въведените правилно данни и да се редактират само въведените неправилно. Ще се наложи въвеждане на всичко отново.

А сега да се занимаем с планирането на входните данни. Като начало ще напишем програмата, която да обработва данните само за няколко ученика и за част от изучаваните предмети. Когато се създава приложение, в което ще се въвеждат много данни, програмата може да се направи така, че отначало да работи с по-малко данни, а след това да се направят поредица от малки изменения и допълнение. И така, стъпка по стъпка, да се получи пълната функционалност. Затова ще направим приложението да работи за 7 ученика и 7 предмета. Когато програмата е готова и работи перфектно, можем да я настроим да работи с реални данни – 26 ученика и 15 предмета.

Естествено хранилище на данните за успеха на numStud ученици по numDisc предмета е двумерен масив с numStud реда и numDisc колони. Затова ще ни е необходима такава графична компонента, в която да можем да съхраняваме, въвеждаме и визуализираме данните така, както в двумерен масив.

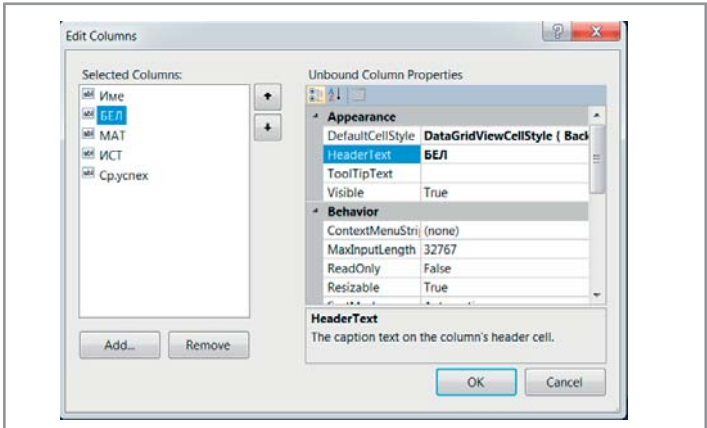
В резултат от обработките програмата ще показва броя на различните оценки по всеки от предметите, както и средния успех за всеки предмет. Затова и за визуализация на резултатите също ще бъде удобна компонента която показва масив с numDisc реда и 6 колони (по една за оценките „слаб“, „среден“, „добър“, „много добър“ и „отличен“, както и една за средния успех). Ако подберем подходяща такава компонента, във формата, от която ще направим прозореца на програмата, ще трябва да поставим две такива компоненти. Една вляво, в която да се въвеждат данните и една вдясно, в която ще изобразяваме статистиките.

Компонентата dataGridView

Компонентата, която ни е необходима се нарича `dataGridView`. Тя представя на екрана двумерна таблица от *клетки* (екземпляри на класа `dataGridViewCell`), всяка от които може да съдържа произволна стойност, за разлика от двумерния масив, в който всички стойности са от един и същ тип. Мястото на клетката в таблицата се определя от номерата на стълба и реда, в които се намира. Забележете още една разлика с двумерния масив, където при идентифицирането на елементите първо задаваме реда, а след това – колоната. Така например клетката в ред `i` и стълб `j` на компонентата `grid1` от тип `dataGridView` указваме като `grid1[j,i]`.

Съдържанието на клетката се задава в свойството Value. Така например, с оператора `grid1[j,i].Value = "Информатика";` задаваме нова стойност на клетката в стълб *j* и ред *i*, а с оператора `string s = (string) grid1[j,i].Value;` или `string s = grid1[j,i].Value.ToString();` присвояваме на променлива от тип `string` стойността на клетката в стълб *j* и ред *i*.

Вътрешно, таблицата е организирана като колекция от *колони* (свойството Columns). За да редактираме колоните на колекцията, трябва да отворим диалоговия прозорец Edit Columns (Фиг. 1). Със стрелките нагоре и надолу може да се промени позицията на маркираната колона. Изтриване на колона става с щракване върху бутона Remove, а нова колона се добавя с щракване на бутона Add..., при което се отваря диалогът Add Column. В полето Header text се изписва наименованието на новата колона.



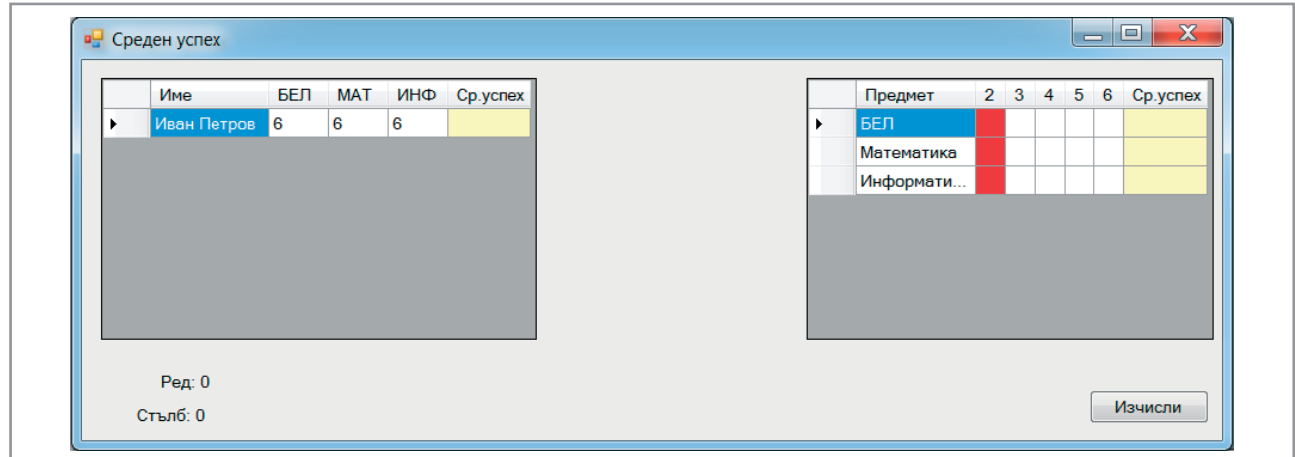
Фиг. 1.

Реализация на проекта

За ваше улеснение, във файла Paralelka1.sln от папка Informatika9-10\Razdel_4 ще намерите готова форма за прозореца на приложението (Фиг. 2). В нея, както планирахме, има две компоненти dataGridView. В едната – DataGrades, ще въвеждаме имената на учениците и оценките им, в другата – DataSpr, ще се извеждат справките (статистиките) по предмети.

В готовата форма са предвидени възможности за въвеждане на имената на трима ученици и оценките им по три предмета, като тези 12 полета могат да се редактират:

```
static int numStud = 3, numDisc = 3;
```



Фиг. 2.

При щракване в някоя от тези клетки, в двата етикета под таблицата се изписват реда и стълба на активната клетка. За целта е прихванато събитието CellClick на DataGrades. Номерата на реда и колоната на клетката се вземат от свойствата RowIndex и ColumnIndex на обекта е от класа DataGridViewCellEventArgs, в който се намират параметрите на събитието CellClick:

```
private void dataGrades_CellClick(object sender,
    DataGridViewCellEventArgs e)
```

```

{ int numRows = e.RowIndex;
  int numCol = e.ColumnIndex;
  label1.Text = "Ред: " + numRows.ToString();
  label2.Text = "Колона: " + numCol.ToString();
}

```

Работа с компютър

В конструкцията на формата е включен програмен код, с който във формата се нанасят името и оценките на първия ученик:

```

dataGrades.Rows.Add(numStud);
dataGrades[0, 0].Value = "Иван Петров";
dataGrades[1, 0].Value = 6;
dataGrades[2, 0].Value = 6;
dataGrades[3, 0].Value = 6;

```

Първата ви задача е да напишете код с който да добавите имената и оценките на още два ученика.

След изпълнението на следващия фрагмент:

```

DataSpr.Rows.Add(numDisc);
DataSpr[0, 0].Value = "БЕЛ";
DataSpr[0, 1].Value = "Математика";
DataSpr[0, 2].Value = "Информатика";

```

в компонентата DataSpr се извеждат имената на трите учебни предмета. Добавете код за извеждане на останалите предмети.

Сега ще трябва да напишете кода за обработката на събитието Click на бутона button1 (с надпис Изчисли). За изчисляването на средния успех на всеки ученик е необходимо да се използва цикъл, в който да се обхождат всички редове (броя на учениците numStud) от компонентата DataGrades. За всяка стъпка на този цикъл в променливата sum се събират стойностите на клетките от текущия ред, т.е. оценките на текущия ученик по всички предмети, което става с вложен цикъл по броя на предметите numDisc. Всяка стойност първо се превръща в низ, а след това се преобразува в числова. Това е така, защото клетките на DataGrades са от общия тип object, което не позволява тяхната директна обработка. Накрая остава да се присвои на последната клетка от текущия ред средния успех на ученика, който се намира като се раздели променливата sum на броя на предметите numDisc. Тъй като това може да е дробно число, то трябва да се използва преобразуване на типа на променливата sum в реален тип чрез добавянето на (double) пред нейното име. За закръгляне на дробно число с точност два знака след десетичната точка се използва методът Round на класа Math:

```

for (int i = 0; i < numStud; i++)
{ int sum = 0;
  for (int j = 1; j <= numDisc; j++)
  sum += int.Parse(dataGrades[j, i].Value.ToString());
  double averageStud = (double)sum / numDisc;
  dataGrades[numDisc+1, i].Value = Math.Round(averageStud, 2);
}

```

Аналогично се намира и средния успех по предмети. Различното тук е, че компонентата DataGrades се обхожда по стълбове и затова външният цикъл е с управляваща променлива, която е ограничена от броя на предметите numDisc, а вътрешният цикъл се изпълнява толкова пъти, колкото е броят на редовете numStud. За всеки предмет се намира броят на двойките, тройките, четворките, петиците и шестиците в пет различни целочислени променливи, след което този брой се записва в съответните клетки на компонентата DataSpr. Изчислява се средния успех за предмета, като отново се извършва преобразуване на цялото число в дробно и закръгляне до втория знак след десетичната точка. Накрая, полученият резултат се записва в съответната клетка от последния стълб на компонентата DataSpr.

```

for (int j = 1; j <= numDisc; j++)
{ int num2 = 0, num3 = 0, num4 = 0, num5 = 0, num6 = 0;
  for (int i = 0; i < numStud; i++)

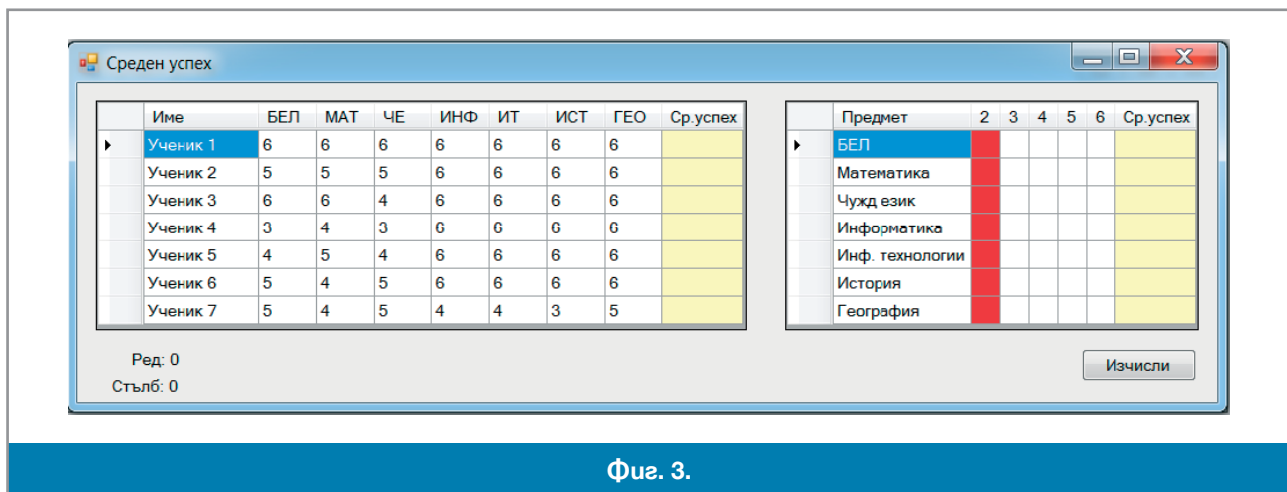
```

```

{ if (dataGrades[j, i].Value.ToString() == "2") num2++;
  if (dataGrades[j, i].Value.ToString() == "3") num3++;
  if (dataGrades[j, i].Value.ToString() == "4") num4++;
  if (dataGrades[j, i].Value.ToString() == "5") num5++;
  if (dataGrades[j, i].Value.ToString() == "6") num6++;
}
DataSpr[1, j-1].Value = num2;
DataSpr[2, j-1].Value = num3;
DataSpr[3, j-1].Value = num4;
DataSpr[4, j-1].Value = num5;
DataSpr[5, j-1].Value = num6;
double avrDisc=(double)(num2*2+num3*3+num4*4+num5*5+num6*6);
avrDisc = avrDisc / numStud;
DataSpr[6, j - 1].Value = Math.Round(averageDisc,2);
}

```

Ако довършите версията на програмата, която да работи за седем ученика и седем предмета ще получите видът на формата от *Фиг. 3*:



Фиг. 3.

Речник

do	ду	прави
for	фор	за
items	айтъмс	елементи
while	уайл	докато

В живота много често се налага да работим с текстове. Затова обработката на текстове е поне толкова важна, колкото и обработката на числови данни. Ако вземем за пример личните данни на един човек ще видим, че по-голямата част от тях са текстови – трите имена; мястото на раждане; името на училището, в което е учил; университетът, който е завършил; фирмите, в които е работил и т.н. За работа с текстове са създадени специални текстообработващи програми, някои от които познаваме от уроците по Информационни технологии. Затова, всеки език за програмиране разполага с възможности за работа с текстови данни.

Типът char

В езика C# знаците се представят с поредния си номер в таблицата Unicode, в която има $2^{16} = 65536$ знака. Стандартът Unicode е създаден в края на 80-те и началото на 90-те години, с цел съхраняването на текстови данни на различни езици. Да напомним, че предшественикът му, таблицата ASCII, позволява записването на едва 128 (в 7-битова версия) или 256 знака (в 8-битова версия). За съжаление, това не удовлетворява съвременните изисквания, особено публикуването в Интернет – тъй като в 128 позиции могат да се поберат само цифрите, малките и главни латински букви и някои специални знаци, а в 255 – и кирилицата, но това е всичко. Ако се наложи работа с текст на кирилица, латиница и някаква трета азбука, например, 256 знака са крайно недостатъчни. Ето защо все по-често използваме 16-битовата кодова таблица Unicode.

В езика C# знаците съхраняваме в променливи от типа `char` – беззнаков целочислен тип. В променлива от тип `char` можем да запишем само един знак. Константите от типа `char` се записват, като съответният знак, когато го има на клавиатурата, се постави между два апострофа. В противен случай между два апострофа се поставя кода на знака предшестван от `\`. Например `'\1040'`.

Работа с компютър

По-долу е дадена програмата `AsciiToUnicode`, която извежда знаците от таблицата Unicode на групи от по 256, по 16 знака на ред, като редът започва с кода на първия от 16-те знака:

```
static void Main(string[] args)
{
    char c; int i,j,k,l;
    for (j = 0; j < 65536; j = j + 256)
    {
        for (k = 0; k <= 15; k++)
        {
            l = j + 16*k;
            Console.Write("{0,5} ", l);
            for (i = 1; i <= 15; i++)
            {
                c = (char)i;
                Console.Write("{0} ", c);
            }
            Console.WriteLine();
        }
        Console.WriteLine("Натиснете клавиш ...");
        Console.ReadLine();
    }
}
```

Въведете програмата и направете от нея конзолно приложение. Опитайте се да намерите местата

(кодовете) на буквите от кирилицата. Не се учудвайте, че за огромното болшинство от знаците, програмата показва въпросителна, вместо съответния знак. Просто дайвърът на конзолата във вашия компютър не е в състояние да изрисува графиките на всички 65536 знака на таблицата Unicode.

Забележете, че за да обърнем целочислената променлива `i` в стойност от типа `char` се налага да използваме операцията за явна смяна на типа на стойност: `(<mun>) <израз>`, резултатът от която е стойността на израза, преобразувана в указания тип.

Типът `string`

За представяне на текстови данни с много знаци – *низове* – може да се използват **масиви от тип `char`**. В някои езици за програмиране това е единственият начин. Отсъствието на отделен тип за представяне на низовете създава много трудности на неопитните програмисти. Затова в езика `C#` е въведен такъв тип. В този урок ще си припомним нещата, които знаем за типа `string` и ще научим още за него.

Променливите от тип `string` в `C#`, всъщност, не съдържат самия низ, а указание къде се намира той в паметта. А там където е разположен низът, той е представен по единствения възможен за компютъра начин – като масив от тип `char`.

Както другите променливи, и променливите от тип `string` трябва да се инициализират, преди да бъдат използвани. Инициализацията може да стане в оператора за деклариране, като след името на променливата се постави равенство и инициализиращата константа от тип `string`. Да напомним, че константи от тип `string` в `C#` се заграждат в кавички. Инициализацията може да стане с оператор за присвояване или пък да се остави на потребителя да въведе желанието от него низ от клавиатурата. Да напомним, че методът `Console.ReadLine()` въвежда това, което потребителят изписва на клавиатурата, като стойност от типа `string`. Извеждането на низове става с метода `Console.WriteLine()`.

Естествена операция с низове е *сливането (конкатенацията)* на два низа. Конкатенацията на низовете α и β е низът $\alpha\beta$. В ирази с низове, обаче, могат да участват и числови стойности, тъй като всяка числова стойност може да се представи с десетични цифри, точка и знаците плюс и минус. Когато е включено в израз с низове, всяко число се преобразува в низа, който е десетичното му представяне. От изученото до момента знаем и още една операция с низове, реализирана с методите `Parse(<низ>)` на класовете, съответни на числовите типове. Тези методи извършват обратни преобразования на описаното по-горе – от низ в число. Ще илюстрираме работата с низове със следната програма, в която всички елементи са ни добре познати:

```
static void Main(string[] args)
{ // Деклариране на променливи
  string s1; int i;
  string s2 = "Здравей ", s3 = " дни.";
  // Въвеждане на данни
  Console.Write("Въведи име:");
  s1 = Console.ReadLine();
  Console.Write("Въведи възраст:");
  i = 365 * int.Parse(Console.ReadLine());
  // Извеждане
  Console.Write(s2 + s1 + ".\n");
  Console.WriteLine("Ти си на " + i + s3);
}
```

Дължина на низ и гостът `go` знаците

Както вече споменахме, всеки низ се съхранява в паметта като масив от тип `char` и броят на знаците в този масив е неговата *дължина*. Дължината на низа е свойство на класа `string` с името `Length`. Така например дължините на инициализираните в току що разгледаната програма низове `string s2 = "Здравей"` и `s3 = "дни."`; е `s2.Length = 8` и `s3.Length = 4`.

Тъй като всеки от знаците на низа е елемент на масива, той има *позиция* в низа и тя е индексът му в масива. Така, ако `s.Length` е `n`, то `s[0]`, `s[1]`, ..., `s[n-1]` са променливи от тип `char`, всяка от

които съдържа поредния знак на низа. Например, за низа `s2 = "Здравей "` имаме `s[0]` е `'З'`, `s[1]` е `'д'`, и т.н., а `s[7]` е `' '`. Обърнете внимание на една важна особеност: посредством индексване на променливата от тип `string` може да се използват отделните знаци на низа, но не може да се променят! Т.е. опитът да се присвои нова стойност, например `s[7] = '!'` ще доведе до грешка още при компилиране на кода.

Работа с компютър

Задача. Напишете конзолно приложение, в което се въвежда от клавиатурата число, цифрите на което са размесени с нецифри. Програмата трябва да „почисти“ въведения низ от знаците, които не са цифри и да изведе в конзолата числото.

ПРИМЕР: **Вход:** `45h(5h1_**21` **Изход:** `45521`

Решение: Ще въведем написаното на клавиатурата в променлива `a` от тип `string`, ще обходим променливата знак по знак (в цикъл управляван от индексната променлива `i = 0, 1, ..., a.Length-1`) и ще отделяме знаците които са десетични цифри. За да бъде един знак десетична цифра, трябва кодът му да е между кодовете на `'0'` и `'9'`. Всяка намерена цифра ще сливаме в края на друга променлива от тип `string`, в която първоначално сме поставили празен низ.

Програма:

```
static void Main(string[] args)
{
    int i;
    Console.Write("Въведете данните:");
    string a = Console.ReadLine();
    string b = "";
    for (i = 0; i < a.Length; i++)
        if (a[i] >= '0' && a[i] <= '9')
            b = b + a[i];
    Console.WriteLine("Числото е: " + b);
}
```

Въведете програмата и създайте от нея конзолно приложение `digits`. Компилирайте я и проверете работоспособността ѝ.

Въпроси и задачи

1. Напишете конзолно приложение, което въвежда низ и извежда броя на главните букви в него – както латински, така от кирилицата. **Упътване:** Формулирайте внимателно условието един знак да е главна буква.

ПРИМЕР: **Вход:** `c5h(5H1Б_Z*u1WЮ` **Изход:** `5`

2. Напишете конзолно приложение, което въвежда низ и проверява дали е *палиндром*, т.е. от ляво на дясно и от дясно на ляво се чете по един и същ начин. Например, `madam` и `001100` са палиндроми. Ако въведеният низ е палиндром, програмата трябва да изведе `Yes`, а ако не е – `No`.

ПРИМЕР 1: **Вход:** `madam` **Изход:** `Yes`

ПРИМЕР 2: **Вход:** `madaam` **Изход:** `No`

3. Напишете конзолно приложение, което въвежда два низа и проверява дали са еднакви. Ако въведените два низ са равни, програмата трябва да изведе `Yes`, а ако не са – `No`. **Упътване:** Първо сравнете дължините на низовете!

ПРИМЕР 1: **Вход:** `madam` **Изход:** `No`

`madaam`

ПРИМЕР 2: **Вход:** `001100` **Изход:** `Yes`

`001100`

В този урок ще разгледаме някои операции, които се използват при обработката на низове. Те са оформени като методи на класа `string`.

Преминаване към главни и малки букви

За преобразуване на всички букви на един низ в малки се използва методът `ToLower()`, а за преобразуване на всички букви в главни – методът `ToUpper()`. Например, ако `s = "Георги"`, след изпълнението на оператора `s1 = s.ToLower();` съдържанието на низа `s1` ще бъде "георги". А след изпълнението на оператора `s1 = s.ToUpper();` съдържанието на низа `s1` ще бъде "ГЕОРГИ".

Работа с компютър

Задача 1. Всъщност нашите знания ни позволяват сами да напишем програмен фрагмент, който преобразува зададен низ в нов низ, заменяйки всички малки букви в зададения низ с главни. За да решим тази задача, нека се върнем към програмата `AsciiToUnicode`, която написахме в предишния урок и да разгледаме кодовете на главните и малките латински букви. Кодът на латинската буква 'A' е 65, а на малката латинска 'a' – 97. За 'B' и 'b' кодовете са съответно 66 и 98, за 'C' и 'c' – 67 и 99, и т.н. Разликата между кодовете на голямата и малката латинска буква е винаги е $32 = 'B' - 'b'$. Ако проверим със същата програма съотношението на кодовете на голямата и малката буква в кирилицата ще забележим че там разликата също е постоянна и е равна на $'Б' - 'б' = 32$ (неслучайно сме избрали за решението на задачата вторите букви на двете азбуки, защото графиките на първите букви на двете азбуки съвпадат и е лесно да се сбъркат една с друга). Затова решението е да обходим с цикъл зададения низ и ако поредната буква е малка латинска да увеличим кода и с $'B' - 'b'$, а ако е малка буква на кирилицата – да увеличим кода ѝ с $'Б' - 'б'$.

Програма:

```
static void Main(string[] args)
{ Console.WriteLine("Въведете низ:");
  string s = Console.ReadLine();
  string t = "";
  for(int i=0;i<s.Length; i++)
  { if(s[i]>= 'a'&& s[i]<='z')
    t = t + (char)(s[i] + 'B' - 'b');
    else
      if(s[i]>='а'&& s[i]<='я')
        t = t + (char)(s[i] + 'Б' - 'б');
      else t = t + s[i];
  }
  Console.WriteLine(t);
}
```

Въведете програмата и направете от нея конзолно приложение. Проверете правилността с един низ, в който непременно да има малки букви на латиницата и кирилицата.

Задача 2. Напишете конзолно приложение, което преобразува зададен низ в нов низ, заменяйки всичко големи букви на латиницата и кирилицата с малки.

Търсене на низ в група низ

Друга често срещана задача при обработката на низове е да се провери дали един низ p (наричан *образец*) се среща в друг низ t (наричан *текст*), т.е. всички знаци на образаца да се срещат в текста един след друг, така както са в образаца. В такъв случай казваме, че p е подниз на t . В езика C# за целта се използват методите `IndexOf(<образец>)` и `LastIndexOf(<образец>)`. Приложен към *<текст>*, всеки от тези методи връща или неотрицателно число, показващо от коя позиция *<образец>* се среща в

<текст> за първи път, като методът `IndexOf` обхожда низа отляво-надясно, а `LastIndexOf` – отдясно-наляво. Ако <образец> не е подниз на <текст>, тогава и двата метода връщат стойност `-1`.

Например, ако стойността на променливата `t` от тип `char` е "алабалиница", тогава:

- ❖ след изпълнение на `int d = t.IndexOf("ала");` стойността на `d` ще бъде `0`;
- ❖ след изпълнение на `int d = t.LastIndexOf("ала");` стойността на `d` ще бъде `4`;
- ❖ а след изпълнение на `int d = t.IndexOf("aaa");` и `int d = t.LastIndexOf("aaa");` стойността на `d` ще бъде `-1`.

Всеки от двата метода може да бъде извикан и с втори аргумент, показващ от коя позиция в низа да започне търсенето. Например, ако

```
string str = "C# Programming Course";
```

тогава `str.IndexOf("r");` ще върне `4`, но `str.IndexOf("r",5);` ще върне `7`, а `str.IndexOf("r",8);` ще върне `18`.

Работа с компютър

Задача 3. Напишете конзолно приложение, което намира броя на всички срещания на зададен образец в зададен текст.

Решение: Ще използваме втория аргумент на метода `IndexOf()`. При първото търсене ще извикаме метода с втори аргумент `0`, за да намери първото срещане на низа. След това трябва да извикаме метода отново и отново, като всеки път във втория аргумент задаваме начало с `1` по-голямо от позицията, където е започвало последното срещане на образаца. Естествено е такъв цикъл, при който стъпката се мени по случаен начин да реализираме с оператора `while`. В променливата `index` ще помним позицията, от която образацът е намерен при последното търсене.

Програма:

```
static void Main(string[] args)
{
    Console.WriteLine("Въведете текст:");
    string t = Console.ReadLine();
    Console.WriteLine("Въведете образец:");
    string p = Console.ReadLine();
    int index = t.IndexOf(p), br = 0;
    Console.WriteLine("{0} се среща: ", p);
    while (index != -1)
    {
        Console.WriteLine("на позиция: {0}", index);
        br++; index = t.IndexOf(p, index + 1);
    }
    Console.WriteLine("Общо: {0} срещания", br);
}
```

Извличане на част от низ

В практиката често се налага да се извличаме подниз от даден низ. За целта трябва да се укаже от коя позиция започва поднизът и неговата дължина или пък от коя позиция започва поднизът и в коя завършва. За целта езикът `C#` предоставя методът `Substring(<начало>, <дължина>)` с два аргумента – позицията на първия елемент на поднизата и дължината му.

Следният програмен фрагмент:

```
string s = "Плевен", s1;
```

```
Console.WriteLine(s);
s1 = s.Substring(1, 3);
Console.WriteLine(s1);
```

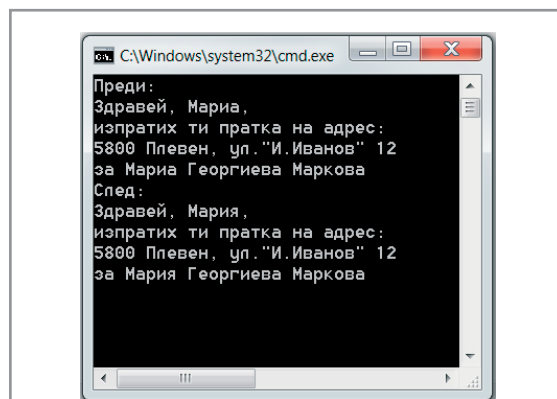
ще завърши с присвояването на променливата `s1` поднизът с дължина `3`, започващ в позиция `1` на низа в променливата `s`, т.е. "лев" (Фиг. 1).

Тази операция също е често срещана в практиката. Например в дълъг текст е въведено грешно името на човек – Мария вместо Мария, например, и трябва да бъде заменено всяко грешно срещане с правилното.

Използва се методът `Replace(<търсен низ>,<заменящ низ>)` с два аргумента – низът, който трябва да се замени и низът, с който ще бъде заменен. В резултат, методът построява нов низ, в който всички срещания на <търсен низ> в текста се заместват с <заменящ низ>. Например:

```
static void Main(string[] args)
{
    string s = "Здравей, Мария,\n"+
        "изпратих ти пратка на адрес:\n"+
        "5800 Плевен, ул.\"И.Иванов\" 12\n"+
        "за Мария Георгиева Маркова";
    string s1;
    Console.WriteLine("Преди:");
    Console.WriteLine(s);
    s1 = s.Replace("Мариа", "Мария");
    Console.WriteLine("След:");
    Console.WriteLine(s1);
    Console.ReadLine();
}
```

Обърнете внимание на факта, че операторът `s.Replace("Мариа", "Мария")`, сам по себе си не променя претърсвания низ а изработва нов. Затова построеният от метода низ трябва да се присвои на друга променлива. Забележете също, че се заменят всички поднизове, а не само първия! Резултатът от изпълнение на програмата е показан на *Фиг. 2*. Напомняме, че управляващият знак `\n` предизвиква преминаване на нов ред.



Фиг. 2.

Въпроси и задачи

1. Разгледайте кода от *Фиг. 3*. В резултат на изпълнението му, на екрана ще се отпечатаат три числа, всяко на отделен ред. На кой ред програмата ще изведе отрицателно число? Защо?
2. Колко ще е стойността на `d1` след изпълнение на програмата от *Фиг. 3*, ако изменим декларацията на `s` в:
`s = "e-mail of Ivo Kolev:kolev@kolev.com";`?
3. Колко ще е стойността на `d3` след изпълнение на програмата от *Фиг. 3*, ако изменим декларацията на `s` в:
`s = "e-mail of Ivo Kolev:kolev@kolev.bg\"";`?

- 4. Въведете кода от *Фиг. 3*. Компилирайте го и се убедете в работоспособността му. Проверете правилността на отговорите на въпросите от **Задачи 1, 2 и 3**, като промените програмата по указания начин. Променете `s1` така, че стойност на променливата `d1` да е 0 след изпълнението на програмата.

```
static void Main(string[] args)
{
    string s,s1,s2;
    s = "Е-мейлът на Иво Колев e kolev@kolev.com";
    int i, d1, d2, d3;
    s1="kolev", s2="KOLEV";
    d1 = s.IndexOf(s1);
    Console.WriteLine(d1);
    d2 = s.LastIndexOf(s1);
    Console.WriteLine(d2);
    D3 = s.LastIndexOf(s2);
    Console.WriteLine(d3);
}
```


Фиг. 3.

5. Изберете такива стойности за променливите p и q, че при изпълнение програмният фрагмент


```
string s = "Езикът C# е обектно-ориентиран";
Console.WriteLine(s);
s1 = s.Substring(p, q);
Console.WriteLine(s1);
```

 да се изведе C#.
6. След изпълнение на програмния фрагмент


```
s = "Георги Георгиев";
s1 = s.Replace("Георги", "Петър");
```

 стойността на променливата s1 ще бъде:
 - а. "Георги Георгиев" б. "Петър Георгиев" в) "Петър Петърев".
7.  Даден е низ, който е пълно име на файл – пътят от корена до съдържащата файла папка и собственото име на файла. Напишете програма, която да въвежда пълното име и извежда собственото име на файла.
ПРИМЕР **Вход:** d:/Izkustva/klas_9/Urok_01/Image1.jpg.
 Изход: Image1.jpg

50
51

Проект „Операции с низове“

Загание

Да приложим наученото досега, за разработване на Windows приложение, в което да могат да се извършват различни операции с низове:

- ❖ преобразуване на низ, замествайки главните букви с малки и обратно;
- ❖ намиране броя на срещанията на образец в зададен текст;
- ❖ извличане на част от низ;
- ❖ замяна на срещанията на подниз с друг низ.

План

В прозореца на програмата, въвеждането на аргументите на операциите и извеждането на получения резултат ще става в текстови кутии. Изборът на операция ще става от група радиобутони, а стартирането на операцията – с натискане на бутон.

Какви компоненти задължително трябва да имаме в прозореца? Две текстови кутии ще са необходими за първата операция – една, в която ще се въвежда зададения низ и една за извеждане на резултата. За втората операция също ще ни трябват две текстови кутии – за текста и за образца, а броят на срещанията ще показваме като етикет.

За третата операция ще са необходими четири текстови кутии – една за зададения низ, по една за началото и дължината на извличания подниз и една за показване на резултата. Толкова текстови кутии ще са нужни и за четвъртата операция – една за зададения текст, една за образца, който се търси в текста, една за низа, с който образецът ще бъде заменян и една за показване на резултата.

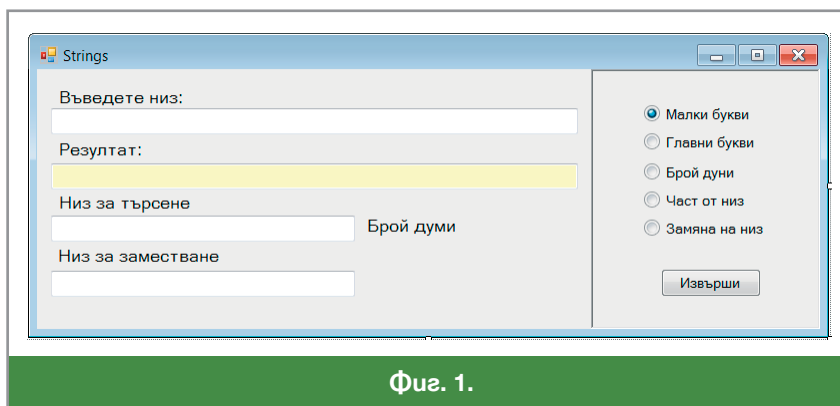
И така, при проектиране на формата в нея ще поставим:

- ❖ четири компоненти TextBox;
- ❖ една компонента Label ще е необходима за резултата на втората операция и още четири за надписи на четирите текстови кутии;
- ❖ група от пет компонентни RadioButton (за първата операция всъщност ще ни трябват два радио бутона – един за замяната на малките с главни и един за обратната замяна);
- ❖ една компонента Button за стартиране на избраната операция.

В папката Informatika9-10\ Razdel_5 ще намерите файла Strings.sln. За ваше улеснение, във формата са добавени необходимите компоненти (Фиг. 1), а в програмната част и обработката на радиобутоните.

На последно място в плана трябва да уточним някои подробности за функционирането на програмата – как ще работи това приложение, кои събития ще се обработват и т.н. Ето поредицата от действия които ще извършва потребителят и какво трябва да прави програмата:

- ❖ Потребителят избира операция, натискайки някой от радиобутоните.
 - ❖ Обработка се събитие Click на натиснатия радиобутон, като във формата се визуализират само тези от текстовите кутии, които са нужни за извършване на избраната операция, както и съответните им етикети.
 - ❖ Потребителят въвежда аргументите на операцията в съответните текстови кутии.
 - ❖ Потребителят натиска бутона за стартиране на избраната операция.
 - ❖ Обработка се събитие Click на бутона за стартиране на избраната операция.
- От стъпките в процедурата, програмистка работа е необходима само във втората и петата стъпка.



Фиг. 1.

Избор на операция

Обработката на събитието Click на натиснатия радиобутон ще се състои в това, да се подготви екранната форма за въвеждане на аргументите на избраната операция. В зависимост от това, кой радиобутон е натиснат, някои от компонентите ще трябва да се направят видими, а други – да се скрият. Трябва да се променят и текстовете в някои от етикетите.

За целта ще се съставим таблица, от която да се вижда, коя компонента в какво състояние трябва да бъде след натискане на всеки от радиобутоните (Фиг. 2). Първата колона на таблицата описва вклю-

Вид	малки главни	главни букви	брой срещания	подниз	замяна
Име					
TextBox TNiz	true	true	true	true	true
Label LRez	true	true	false	false	False
TextBox TNizRez	true	true	false	false	false
Label LDuma1	false	false	true/ Търси	true/ Начало	true/Замени с
TextBox TDuma1	false	false	true	true	true
Label LDuma2	false	false	false	true/Дължина	true/Замени с
TextBox TDuma2	false	false	true	true	true
Label LBroy	false	false	true	false	false

Фиг. 2.

чените във формата компоненти. Горے в клетката е типът на компонентата, а под него – името ѝ. Всяка от останалите колони е отделена за една от операциите.

В клетката на компонентата *K* и операцията *O* е стойността на свойството `Visible` – `true` за да се вижда или `false` за да не се вижда компонентата. На два от етикетите ще се наложи да се променя свойството `Text` в зависимост от това коя операция се изпълнява и съответния текст също е показан в таблицата.

Ще използваме следните променливи:

```
string Niz, NizRez, Duma1, Duma2;  
int Nachalo, Daljina, Broy;
```

`Niz` е променливата, в която съхраняваме низа, който ще се обработва. Въвежда се от потребителя в текстовата кутия `TNiz`. Резултатът от операцията ще получаваме в променливата `NizRez`, а ще показваме в текстовата кутия `TextNizRez`. `Duma1` е низ, който има „двойна“ роля – в една от операциите ще е низът който се търси, а в друга – който се търси и замества. Въвежда се от потребителя в текстовата кутия `TDuma1`. `Duma2` е низът, с който ще се замества. Въвежда се от потребителя в `TDuma2`.

В цялата променлива `Nachalo` ще бъде позицията от който ще започне извличането на подниза. Въвежда се от потребителя в `TDuma1`. В променливата `Dylghina` ще държим дължината на подниза който ще се извлича. Въвежда се от потребителя в `TDuma2`. В последната цяла променлива `Broy` ще намираме броя на срещанията на търсената дума.

За ваше улеснение обработката на събитието `Click` на радиобутоните е въведена в кода на програмата. Тя проверява кой от радиобутоните е активен и визуализира компонентите, необходими за операцията. Избраният от групата радиобутони установяваме по това, че свойството му `Checked` има стойност `true`. Съответният код изглежда така:

```
if (RBMalkiB.Checked)  
{ Подготовка за замяна на големите букви в низа с малки }  
if (RBGlawniB.Checked)  
{ Подготовка за замяна на малките букви в низа с главни }  
и т.н.
```

Извършване на операцията с обработка на изключенията

Вече е ставало дума в предишен урок, че потребителят може, умишлено или не, да допусне грешки при въвеждане на данните. Някои такива грешки могат да доведат до *аварийно завършване* на работата на програмата или до *зацикляне*, в резултат на което програмата, никога няма да завърши сама и трябва да бъде спряна от потребителя. Добре е, когато планираме програмата, да предвидим повечето от тези грешки и да създадем програмата така, че да реагира подобаващо.

Първото, и най-важно условие е, да се въведе низът, който ще се обработва. В променливата `Niz` низът се присвоява от свойството `Text` на текстовата кутия `TNiz`. Затова преди да обработваме събитието `Click` на бутона на операцията, трябва да проверим дали наистина е въведен непразен низ. Ако той е празен, т.е. `TextNiz.Text` е `""` или `TNiz.Length` е нула, няма смисъл да се обработва събитието `Click` на бутона Извърши.

```
Niz = TNiz.Text;  
if (Niz != "")  
{ if (RBMalkiB.Checked)  
  { NizRez = Niz.ToLower();  
    TNizRez.Text = NizR;  
  }  
  else  
  if (RBGlawniB.Checked)  
  { ..... }  
}
```

Тази проверка не е достатъчна за качествената работа на програмата. При различните операции (избора на радиобутоните) потребителят трябва да въвежда различни по тип стойности. Да разгледаме изпълнението на операцията *Извличане на част от низ* и обработката на грешки при нея, защото тук необходимите проверки са най-много. В текстовите кутии за началото на низа и дължината му, трябва да се въведат само цифри и първата проверка е точно такава:

```

try
{
    Nachalo = int.Parse(Duma1);
    Daljina = int.Parse(Duma2);
}
catch (Exception)
{
    // Грешни Duma1 и Duma2
    Nachalo = 0;
    Daljina = 0;
}
finally
{
    .....
}

```

Забележете, че в частта `catch (Exception)` при възникване на грешка се дава стойности `0` на двете променливи и се осигурява, че данните са числа. В частта `finally`, операторите на която се изпълняват във всички случаи, трябва да се направи още една проверка – дали началото и дължината са **допустими числа**. Не бива да започваме да изпълняваме операцията, ако потребителят е задал несъществуващо начало на низа, например `-1`? Или ако е зададен къс низ и по-дълъг подниз. Например, в низ с дължина `4` да се намери подниз с дължина `25`. Затова се налага още една проверка в частта `finally`:

```

if (Nachalo<0 || Dalghina<0 || Nachalo+Dalghina>Niz.Length)
{
    // Подходящо съобщение за грешка
}
else
{
    try
    {
        NizR = Niz.Substring(NomChar, Dylghina);
    }
    catch (Exception)
    {
        NizR = ""; // Грешни NomChar и Dylghina
    }
    finally
    {
        textNizRez.Text = NizR;
    }
}

```

Така трябва да се пише програмата, когато в нея има въвеждане на данни от потребителя, за да не завърши аварийно и да не зацikli по време на изпълнението.

Работа с компютър

Довършете програмата, компилирайте я и проверете работоспособността на всички функции, като задавате както коректни така и грешни входни данни за да се убедите, че не завършва аварийно и не зацikliя.

52

Сравняване на низове

Сравняването на низове се различава доста от сравняването на числа. В този урок ще припомним възможностите за сравняване на низове с операциите за сравняване, които познаваме и ще покажем и други възможности, оформени в езика *C#* като методи на класа `string`.

Сравняване за съвпадение (несъвпадение)

Да припомним тук, че от познатите ни операции за сравняване, върху стрингове са приложими само операцията проверяване за равенство (`==`) и проверяване за неравенство (`!=`).

Ако `string s = "Пловдив"`; и `string s1 = "Пловдив"`; тогава операторът `Console.WriteLine(s == s1)`; ще изведе на конзолата `true`, а ако `string s = "Пловдив"`; и `string s1 = "ПЛОВДИВ"`; тогава ще изведе `false`.

Друга възможност за сравняване на низове за съвпадение/несъвпадение е с метода `Equals(<низ>)` на класа `string`. Когато методът се извиква с един аргумент, се сравняват низът, към който е приложен методът, и низът аргумент. Така например изпълнението на програмния фрагмент:

```
string s = "Пловдив", s1 = "ПЛОВДИВ";  
Console.WriteLine(s.Equals(s1));
```

ще изведе на конзолата `false`.

За игнориране на разликата между малки и главни букви при сравняването, може да се постъпи по различни начини. Например, да се превърнат двата низа само в главни или само в малки букви и едва тогава да се сравнят:

```
string s = „Пловдив“, string s1="ПЛОВДИВ";  
Console.WriteLine(s.ToLower()==s1.ToLower()).
```

или с метода `Equals()`, с втори параметър константата `CurrentCultureIgnoreCase` – атрибут на класа `StringComparison`:

```
using Systems;  
string s = "Пловдив", s1 = "ПЛОВДИВ";  
WriteLine(s.Equals(s1,StringComparison.CurrentCultureIgnoreCase));
```

Азбучно (лексикографско) сравняване

От уроците по Информационни технологии познаваме *лексикографската (азбучната) наредба*, която ни позволява да сравняваме низовете за отношенията по-малко, по-малко или равно, по-голямо и по-голямо или равно. За да сравняваме лексикографски низовете, трябва първо да подредим знаците по големина. За латинската азбука и кирилицата подредването е стандартното – това при което изреждаме буквите когато изучаваме азбуката, а за цифрите – подредването е по големина. Лексикографски сравняваме низовете знак по знак до срещне на първото несъвпадение. Този от двата низа, при когото се е срещнал по малкия в наредбата на знаците е по-малък лексикографски. Например низът **"риба"** е преди низа **"рима"**, а **"12345678"** е преди **"9"**.

За лексикографско сравняване на низове в езика `C#` се използва методът `CompareTo(<низ>)` на класа `string`. Приложен към променливата от тип низ `str1` с аргумент низът в променливата `str2` – `str1.CompareTo(str2)`, методът връща:

- ❖ `-1`, когато е в сила `str1 < str2`;
- ❖ `0` когато е в сила `str1 == str2` (т.е. съвпадат);
- ❖ `1` когато е в сила `str2 < str1`.

Например, ако `str1 = "риба"`, а `str2 = "рима"`, тогава операторът

```
Console.WriteLine(s1.CompareTo(s2));
```

 ще изведе на конзолата `-1`, а операторът

```
Console.WriteLine(s2.CompareTo(s1));
```

 ще изведе `1`.

Внимание ако `str1 = "риба"`, а `str2 = "Риба"`, тогава операторът `Console.WriteLine(s1.CompareTo(s2));` ще изведе на конзолата `-1`. Независимо от това, че в таблицата `Unicode` големите букви (и на кирилицата и на латиницата) предшестват малките, в наредбата на знаците те са подредени обратно.

Правилото, което формулирахме по-горе не дава решение в случая, когато единият низ е *начало* на другия, т.е. всички знаци на първия съвпадат със съответните знаци на втория. Да разгледаме примера:

```
string s1 = "тон", string s2 = "тоника";  
Console.WriteLine(s1.CompareTo(s2));
```

В резултат от изпълнението, на конзолата ще бъде изведена стойността `-1`, т.е. **"тон" < "тоника"**! Когато един низ е начало на друг, тогава по-късият предхожда по-дългия в лексикографската наредба.

Интересен ефект има сравняването на низове съдържащи само цифри. Възможно ли е две променливи да имат стойност `34` и `123`, съответно, а резултатът от сравняването им да е `123 < 34`? Ако променливи са целочислени това не е възможно. Когато обаче са от тип `string`, низът **"123"** предшества низа **"34"** в лексикографската наредба.

Интересно е да се сравнят двата низа `s1 = "Иван Иванов"` и `s2 = "Иван Иванов"`. Разликата на пръв поглед е незначителна – в `s1`, между името и фамилията, има един интервал повече. Затова при сравняване на тези два низа, резултатът ще бъде, че `s1 < s2` ... Обяснението е просто. Интервалът

е с код 32 в таблицата Unicode, което значи, че е преди всички букви. Тъй като от началото до позиция 4 всички знаци на двата низа съвпадат, сравняването ще приключи в позиция 5, където кодът на интервала s1 е по-малък от кода на буквата И в s2 (Фиг. 1).

s1	И	В	А	Н			И	...
s2	И	В	А	Н		И	В	...
Позиция	0	1	2	3	4	5	6	...

Фиг. 1.

Въпроси и задачи

- Какъв ще е резултатът от сравнението на низовете:
 - s1 = "abcd" и s2 = "abc";
 - s1 = "56" и s2 = "243";
 - s1 = "abcd" и s2 = "abcd";
 - s1 = "0056" и s2 = "0243".
- Какво ще се изведе на конзолата след изпълнението на фрагмента от програмата
 - показан на Фиг. 2а;
 - показан на Фиг. 2б?

<pre>string s1 = "Емилия"; string s2 = "Емил"; Console.WriteLine(s1.CompareTo(s2));</pre> <p style="text-align: center;">а.</p>	<pre>string s1 = "обор"; string s2 = "бор"; Console.WriteLine(s2.CompareTo(s1));</pre> <p style="text-align: center;">б.</p>
---	--

Фиг. 2.

- Може ли да се сортира масив от тип string с CompareTo()? Ако не, обяснете защо, а ако може – напишете конзолно приложение за сортиране на масив от низове.
- След изпълнение на следния фрагмент от програмата:


```
string s1 = "Алекс", s2 = "Алекс";
Console.WriteLine(s1==s2);
```

 на конзолата се извежда false. Каква може да е причината?

53

Низове – упражнение

Задача 1. Напишете конзолно приложение, което въвежда два низа от цифри, допълва по-късия от тях с нули отляво, докато станат с еднаква дължина, и извежда получения низ. Ако двата низа са с еднаква дължина, програмата трябва да изведе ОК.

Пример 1:

Вход:	Изход:
20412	00321
321	

Пример 2:

Вход:	Изход:
345	ОК
345	

Задача 2. Напишете конзолно приложение, което извежда по-малкото от две числа, въведени в променливи от тип низ. Дължината на числата е от 1 до 1000 цифри.

Пример 1:

Вход:	Изход:
21342	321
321	

Пример 2:

Вход:	Изход:
345	345
345	

Задача 3. Напишете конзолно приложение, в което се въвежда числото *n* и имената на *n* града, а извежда първия и последния в лексикографската им подредба.

Пример:

Вход: 4
Бургас
Ямбол
Айтос
Варна

Изход: Айтос
Ямбол

Задача 4. Напишете конзолно приложение, което въвежда низ и отпечатва съдържанието му, четено отдясно наляво.

Пример:

Вход: кораб **Изход:** барок

Задача 5. Едно просто *криптиране* (т.е. кодиране на текст така, че да не може да се чете от човек, не познаващ декодиращия алгоритъм) е следното: всяка буква се замества със следващата в азбуката, а буквата я с буквата а. Например, думата **роза**, след криптиране ще стане **спиб**. Напишете конзолно приложение, което въвежда дума и я извежда криптирана.

Пример:

Вход: кола **Изход:** лпмб

Задача 6: Криптирането през една буква наричаме *със стъпка 1*. При криптиране със стъпка 3 всяка буква ще се замени с третата след нея в азбуката: а → г, б → д, ..., ъ → я, ь → а, ю → б, я → в. Напишете конзолно приложение, което по зададена стъпка n ($0 < n < 31$) кодира въведена дума.

Пример:

Вход: 2 **Изход:** грйв
боза

Задача 7: Напишете конзолно приложение, което въвежда низ и го извежда като замества всички цифри в низа със звездички.

Пример:

Вход: Николай от 9а клас има ЕГН 9603124004.

Изход: Николай от *а клас има ЕГН *****.

Задача 8: Напишете конзолно приложение, което въвежда низ съдържащ кръгли скоби и извежда YES ако скобите са поставени правилно и NO в противен случай.

Пример 1:

Вход: (2+3.(4-1))+2

Изход: YES

Пример 2:

Вход: 2+(3+2).5)

Изход: NO

Пример 3:

Вход: (2+1).2)+2)

Изход: NO

Упътване: Не е достатъчно само да проверите дали броят на левите скоби е равен на броя на десните. В **Пример 3** равенството е в сила, но скобите са поставени неправилно. За да решите задачата, нулирайте в началото една променлива s , прибавяйте в s 1 при срещане на лява скоба и изваждайте 1

при срещане на дясна. Ако след поредното изваждане s стане -1 , скобите са поставени неправилно. Колко трябва да е стойността на s в края на алгоритъма, за да бъде изразът правилен. Броенето за Пример 3 е показано в таблицата на *Фиг. 1*.

Низ	(2	+	1)	.	2)	+	2	(
s	1				0			-1			

Фиг. 1.

54 Тест: Низове

1. Даден е низът $s = \text{"Програмиране"}$. Знакът $s[3]$ е:
а) 'о'; б) 'м'; в) 'г'; г) 'а'.
2. Дадени са двата низа $s1 = \text{"1"}$ и $s2 = \text{"2"}$.
След изпълнение на оператора `Console.WriteLine(s1 + s2);` в конзолата ще се изведе:
а) 1; б) 2; в) $s1 + s2$; г) 12.

<code>int n = 12, p; string s1 = "5"; p = s1 + n; Console.WriteLine(p);</code>	<code>s = "Георги"; s2 = s.ToLower(); s3 = s.ToUpper(); Console.WriteLine(s3);</code>	<code>s = "Име:Иван Иванов"; d = s.IndexOf("Иван") + s.LastIndexOf("Иван");</code>
--	---	--

а.

б.

в.

Фиг. 1.

3. След изпълнение на програмния фрагмент от Фиг. 1а на конзолата ще се изведе:
 - а) 12;
 - б) 125;
 - в) 512;
 - г) фрагментът е грешен.
4. След изпълнение на програмния фрагмент от Фиг. 1б на конзолата ще се изведе:
 - а) ГЕОРГИ;
 - б) георги;
 - в) Георги;
 - г) друго.
5. След изпълнение на програмния фрагмент от Фиг. 1в стойността на d ще бъде:
 - а) 4;
 - б) 13;
 - в) 10;
 - г) друг отговор.

<code>s = "Пампорово"; s1 = s.Substring(3, 4); Console.WriteLine(s1);</code>	<code>s = "111 222 121 11111"; s1 = s.Replace("111", "333");</code>
--	---

а.

б.

Фиг. 2.

6. След изпълнение на програмния фрагмент от Фиг. 2а на конзолата ще се изведе:
 - а) поро;
 - б) мпор;
 - в) пор;
 - г) друго.
7. Колко замени ще бъдат извършени при изпълнение на програмния фрагмент от Фиг. 2б:
 - а) 4;
 - б) 2;
 - в) 3;
 - г) 1?

<code>s1 = "23"; s2 = "145"; if (s1 < s2) Console.WriteLine(s1); else Console.WriteLine(s2);</code>	<code>s1 = "кора"; s2 = "корал"; n = s1.CompareTo(s2);</code>
--	---

а.

б.

Фиг. 3.

8. След изпълнение на програмния фрагмент от Фиг. 3а в конзолата ще бъде изведено:
 - а) 23;
 - б) 145;
 - в) 23145;
 - г) друго.
9. След изпълнение на програмния фрагмент от Фиг. 3б стойността на n ще бъде:
 - а) 0;
 - б) -1;
 - в) 1;
 - г) друга.

<code>s1 = "кора"; s2 = "корал"; n = s1.CompareTo(s2) + s2.CompareTo(s1);</code>	<code>s1 = "кораб"; s2 = "кора"; n = s1.CompareTo(s2) + s2.CompareTo(s1);</code>
--	--

а.

б.

Фиг. 4.

10. След изпълнение на програмния фрагмент от Фиг. 4а стойността на n ще бъде:
 - а) 0;
 - б) 2;
 - в) 1;
 - г) друга.
11. След изпълнение на програмния фрагмент от Фиг. 4б стойността на n ще бъде:
 - а) 0;
 - б) 2;
 - в) 1;
 - г) друга.

Компютърна графика

С компютърната графика се срещнахме в уроците по ИТ. От тези уроци знаем как компютърът построява различни графични изображения. Компютърният монитор е правоъгълна таблица от квадратчета – *пиксели*, наречена *растер*. Таблицата може да има повече или по-малко редове и стълбове. Тази характеристика на графичното поле наричаме *разделителна способност*. Колкото по-голяма е разделителната способност, толкова по-добри изображения получаваме.

Всеки пиксел може да „свети“ в един от няколко милиона цвята. Компютърното изображение се получава като за всеки пиксел се избере подходящ цвят. Съществуват два принципно различни способа за създаване и съхраняване във файл на компютърна графика:

- ❖ **Растрерна графика.** Растрерното графично изображение (bitmap) е съставено от цветните стойности на всеки отделен пиксел. Растрерното изображение е обемието и след като веднъж е изработено не може лесно да се промени. При опит да се увеличи или намали, качеството му силно се влошава. Растрерни изображения се създават и обработват, например, с познатата ни от уроците по ИТ програма Paint.
- ❖ **Векторна графика.** Векторното изображение е съставено от отделни графични елементи – части от прави и криви линии, различни фигури и т.н., всеки от които е представен с математическо описание. Затова представянето е с малък обем. Когато векторното изображение трябва да се представи на екрана – математическото описание се трансформира в множество от цветни точки, които изменят съответните пиксели на екрана. Затова векторните изображения се увеличават или намаляват без да се губи качеството им. Векторната графика също ни е позната от работата с програмите на пакета Microsoft Office.

Езикът C# предлага инструментариум за създаване на векторна графика, включваща и текст – класът **Graphics**, дефиниран в пространството от имена **System.Drawing**. Средата включва автоматично в програмния код на Windows Forms Application това пространство. Класът **Graphics** е имплементация на системата за създаване на графични изображения GDI+ (Graphics Device Interface), която е развита версия на WGDI (Windows Graphics Device Interface).

Създаване на Windows приложение с графика

Написването на програма, в която ще създаваме графични изображения, започва с отварянето на приложение с графичен интерфейс. След това трябва задължително да **напишем** в класа **Form1** специален, *защитен* и *предефиниран* (**protected override**), метод за обработка на събитието **OnPaint** с един аргумент – обект **e** от класа **PaintEventArgs**. От свойството **Graphics** на **e** се създава чертожно поле – обект **g** от класа **Graphics**, в който можем да създадем нашето графично изображение:

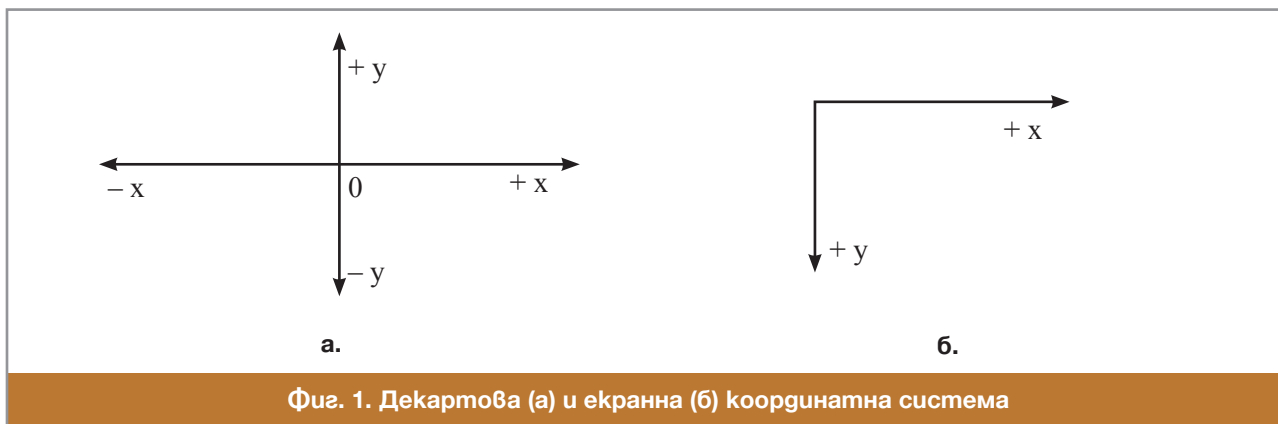
```
protected override void OnPaint(PaintEventArgs e)
{ Graphics g = e.Graphics; }
```

Този метод се изпълнява при отварянето на формата и съобщава на програмата, че вътрешността на екранната форма, няма да се използва за разполагане на компоненти, а за създаване на графика, чрез програмния код написан за **g**. Изчертаването на графичните обекти трябва да стане също в този метод.

Чертожно поле

В математиката представянето на обекти в равнината става в *правоъгълна координатна система*, наричана още *Декартова*, в чест на френския математик и философ Ръне Декарт (1596 г. – 1650 г.). Негова е заслугата за въвеждането на координатна система в равнината и създаването на *аналитичната геометрия*, която е математическата основа на компютърната графика.

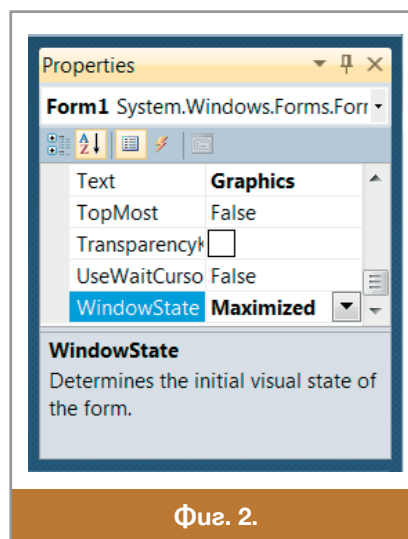
Правоъгълната координатна система (Фиг. 1а) се дефинира от две перпендикулярни прави в равнината, наричани *координатни оси* – *хоризонтална* (или *абсцисна*) ос Ox и *вертикална* (или *ординатна*) ос Oy . Пресечната точка O на двете оси се нарича *начало* на координатната система. Точките на всяка от осите съответстват на реалните числа, като началото е нулата на всяка от осите. На абсцисната ос, положителните числа са надясно от нулата, а отрицателните – наляво. На ординатната ос, положителните числа са нагоре от нулата, а отрицателните – надолу. С въвеждането на координатна система, всяка точка P на равнината се представя с наредена двойка реални числа (x,y) – *координати* на P . Координатата x е реалното число, съответно на ортогоналната проекция на P върху оста Ox , а координатата y – реалното число, съответно на ортогоналната проекция на P върху оста Oy . Началото O на координатната система е с координати $(0,0)$.



Фиг. 1. Декартова (а) и екранна (б) координатна система

При работа с компютър се използва различна координатна система. В компютърната графика равнината е ограничена до т.н. *чертожно поле* – правоъгълна част от екранния raster с r реда и c стълба, която заема вътрешността на екранната форма (Фиг. 1б). Екранната координатна система, за разлика от Декартовата, е *дискретна* – екранните точки са пикселите и координатите им са само цели положителни числа. Пикселът в горния ляв ъгъл на чертожното поле е началото на координатната система – координати $(0,0)$. x -координати са целите числа от 0 до $c - 1$ и растат отляво надясно. y -координатите са целите числа от 0 до $r - 1$ и растат отгоре надолу (тъй като обновяването на изображението на монитора става с обхождане на пикселите отгоре надолу и отляво надясно).

Реалните размери r и c на чертожното поле, зависят от това, колко голям е прозорецът на програмата. Най-голямо чертожно поле ще получим, разбира се, ако максимизираме прозореца. Това може да стане програмно със задаване стойност `Maximized` на свойството `WindowState` на формата (Фиг. 2).



Фиг. 2.

Текущите размери на чертожното поле на формата се съдържат в свойствата `ClientSize.Width` – ширина и `ClientSize.Height` – височина на чертожното поле.

Създаване на писалка

Когато чертаем върху хартия, използваме молив, химикалка, флумастер или друг пишещ инструмент. Така постъпваме и в компютърната графика. Инструментът, който избираме, се характеризира основно с цвета и дебелината на следата, която оставя в чертожното поле. Тези и други характеристики, са обобщени в класа `Pen` (писалка). Затова, преди да започнем изчертаването, трябва да създадем обект от този клас:

```
Pen p = new Pen(<цвет>, <дебелина>); .
```

Цветът е обект от класа `Color` със 140 свойства, всяко от които е някакъв цвят и се избира от списъчна кутия, която се отваря след написване на знака точка след името на класа. Дебелината на линията

е число от тип `float`, и може да считем, че закръглено до цяло число, то означава брой пиксели. Всяка стойност на дебелината по-малка от 1 се приема за дебелина 1. Операторът `Pen p = new Pen(Color.Red, -3);`, например, създава писалка с червен цвят и дебелина 1 пиксел.

Ако искаме да създадем писалка с цвят, който не е измежду включените като свойства на класа `Color`, трябва да използваме статичния метод `Color.FromArgb(<червен>, <зелен>, <син>)`, който задава цвят, базиран на модела RGB (red-green-blue), т.е. чрез интензитетите (цели числа от 0 до 255) на трите образуващи цвята – червено, зелено и синьо:

```
Pen p = new Pen(Color.FromArgb(49, 226, 29), 2);
```

Трите интензитета на избрания цвят могат да се вземат от цветовата палитра на програмата `Paint`, като се отвори палитрата и се избере интересуваният ни цвят.

Фонов цвят на чертожното поле

Фоновият цвят на чертожното поле се задава като стойност на свойството `BackColor`. Друг начин за смяна на фоновия цвят на полето е с метода `Clear` на класа `Graphics`. Той има един параметър, който е обект от класа `Color`:

```
g.Clear(Color.Red); или  
g.Clear(Color.FromArgb(20, 140, 60));
```

Работа с компютър

Ще напишем програма, която отваря екранна форма, трансформира я в чертожно поле и с оператор за цикъл променя многократно цвета на фона с генериран по случаен начин цвят, при което се получава интересен ефект:

```
protected override void OnPaint(PaintEventArgs e)  
{ Graphics g = e.Graphics;  
  int a = 255, freq=10;  
  Random c = new Random();  
  for (int i = 1; i <= 40; i++)  
  { g.Clear(Color.FromArgb(c.Next()%a, c.Next()%a, c.Next()%a));  
    System.Threading.Thread.Sleep(1000 / 10);  
  }  
}
```

Такъв прием се използва при създаването на анимационни ефекти, например. Методът `System.Threading.Thread.Sleep(<време>)`, който ще използваме при създаването на компютърна графика забавя изпълнението на следващия оператор за времето в милисекунди, зададено като параметър. Всяко извикване `Next()` на обекта `c` от класа `Random` пък връща едно случайно число, което превръщаме в интензитет (от 0 до 255) като вземем остатъка му по модул 255.

56
57

Геометрични фигури

Изчертаване на права линия

За изчертаване на права линия между две екранни точки се използва методът `DrawLine` на класа `Graphics`, с параметри писалка и четири цели числа:

```
void DrawLine(Pen p, int x1, int y1, int x2, int y2)
```

Писалката `p` задава цвета и дебелината на линията. Първата двойка числа са координатите на първата точка, а втората двойка числа – на втората. `DrawLine` чертае линия от първата точка до втората точка включително. Например:

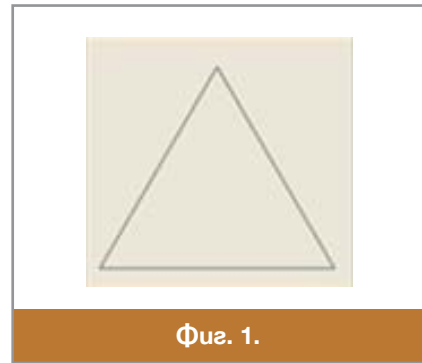
```
Pen p = new Pen(Color.Black, 0);  
g.DrawLine(p, 0, 0, 15, 15);
```

ще оцвети в черно 16-те пиксела с координати (0,0), (1,1), и т.н., до (15,15), така че в чертожното поле те се виждат като оцветена в черно отсечка. Редът на задаването на двете точки няма значение, затова извикването:

```
g.DrawLine(p, 15, 15, 0, 0);
```

чертае същата отсечка. При задаване на съвпадащи две точки DrawLine не чертае нищо.

Задача 1: Напишете програма, която изчертава равностранен триъгълник със страна 200 пиксела. Променете програмата така, че дължината на страната да бъде задавана като стойност на променлива (Фиг. 1).



Фиг. 1.

Решение: Ще решим втората подзадача. За да се изчертата триъгълник, трябва да се изчертаят трите му страни. За целта, трябва да се намерят координатите на върховете му, тъй като те ще бъдат параметри на метода DrawLine. Нека променливите `x1` и `y1` са инициализирани с координатите на долния ляв връх, а променливата `side` с дължината на страната:

```
int x1=200, y1=500, side=200; .
```

За да бъде триъгълникът равностранен, стойностите за координатите на другите два върха трябва да бъдат пресметнати математически. Така за координатите `x2` и `y2` на долния десен ъгъл се получава:

```
int x2 = x1 + side, y2 = y1;
```

Малко по-сложно се намират координатите `x3` и `y3` на третия връх. Неговата абциса съвпада със средата на основата на триъгълника:

```
int x3 = (x1 + x2) / 2; ,
```

а ординатата му се получава от ординатата на левия долен ъгъл, като я намалим с височината на триъгълника. Височината на равностранен триъгълник със страна `side` може да се намери с Питагоровата теорема и е равна на $side \cdot \sqrt{3} / 2$. Затова да пресметнем стойността на височината и да я преобразуваме в целочислен тип, чрез поставянето на `(int)` пред дробния израз, след което да я извадим от `y1`:

```
int h = (int)(side * Math.Sqrt(3) / 2); y3 = y1 - h; .
```

Сега вече сме готови да изчертаем триъгълника, като изчертаем поотделно всяка от трите му страни:

```
g.DrawLine(p, x1, y1, x2, y2);
g.DrawLine(p, x2, y2, x3, y3);
g.DrawLine(p, x3, y3, x1, y1);
```

Ето и целия програмен код събран в тялото на метода `OnPaint`:

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen p = new Pen(Color.Black, 2);
    int side=200, x1=200, y1=500, x2, y2, x3, y3;
    x2 = x1 + side; y2 = y1; x3 = (x1 + x2) / 2;
    int h = (int)(side * Math.Sqrt(3) / 2);
    y3 = y1 - h;
    g.DrawLine(p, x1, y1, x2, y2);
    g.DrawLine(p, x2, y2, x3, y3);
    g.DrawLine(p, x3, y3, x1, y1);
}
```

Класът Point

Тъй като в компютърната графика много често работим с точките на екранната координатна система, в пространството от имена `System.Drawing` е дефиниран класът `Point`. Най-важните свойства на обект от класа са абсцисата `X` и ординатата `Y` на точката. За изчертаване на права линия между две екранни точки се използва версията на метода `DrawLine` в която вместо четири цели координати се задават две точки:

`g.DrawLine(Pen p, Point a, Point b)`. За да стане това трябва предварително да конструираме точките `a` и `b` от координатите им:

```
Pen p = new Pen(Color.DarkOrchid, 2);
```

```
Point a = new Point(123, 24), b = new Point(12, 242);
g.DrawLine(p, a, b);
```

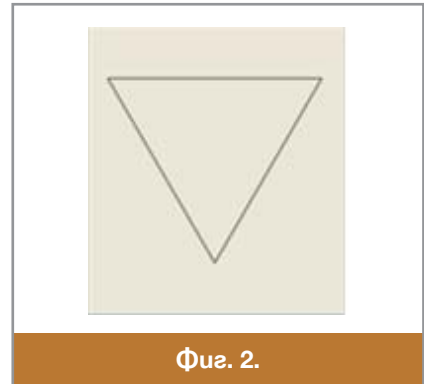
Работа с компютър

1. Експериментирайте с програмния код на решената в урока задача, като промените големината на страната в `side` и началното положение на координатите `x1` и `y1`. Променя ли се при това размерът на триъгълника и неговото местоположение? А остава ли триъгълникът равностранен?

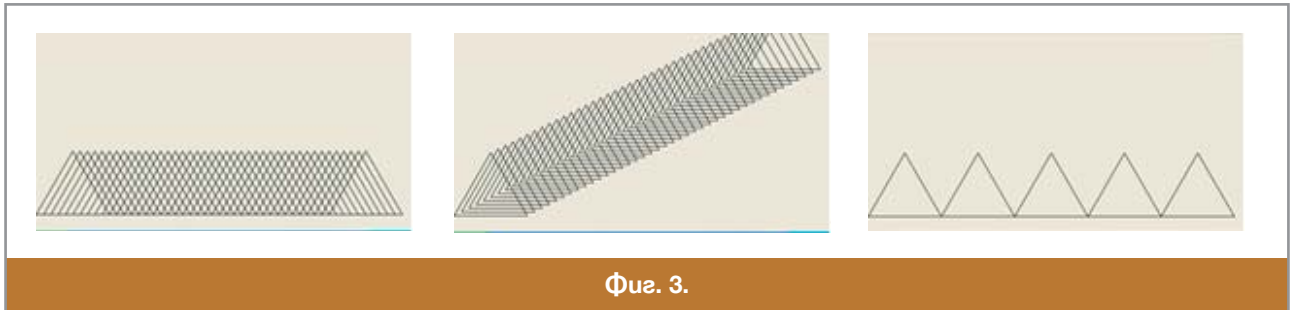
2. Какво трябва да се промени в кода на тази програма, за да се изчертае триъгълникът обрнат надолу (Фиг. 2)?

Упътване. Пресметнете отново ординатата на третия връх.

3. Как бихте направили картинките от Фиг. 3? Сигурно се досещате, че при изчертаването е използван оператор за цикъл! Ето как трябва да се промени кода на програмата от урока, за да се получи втората картинка:



Фиг. 2.



Фиг. 3.

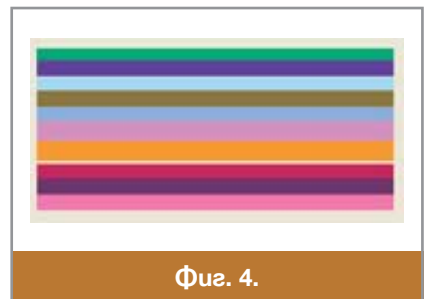
```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen p = new Pen(Color.Black, 2);
    int side = 200, x1 = 0, y1 = 500, x2, y2, x3, y3;
    for (; x1 <= 800; x1 += 20, y1 -= 10)
    {
        x2 = x1 + side; y2 = y1;
        x3 = (x1 + x2) / 2;
        int h = (int)(side * Math.Sqrt(3) / 2);
        y3 = y1 - h;
        g.DrawLine(p, x1, y1, x2, y2);
        g.DrawLine(p, x2, y2, x3, y3);
        g.DrawLine(p, x3, y3, x1, y1);
    }
}
```

Опитайте се сами да направите другите две картинки. Може да пробвате да направите и собствени, като, например, промените в тялото на цикъла и дължината на страната на триъгълника.

4. Напишете програма, която изчертава 10 хоризонтални линии с дължина 500, разположени плътно едно до друга, със случайна дебелина и цвят (Фиг. 4).

Решение:

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Random r = new Random();
    int h = 100, red, green, blue;
    for (int i = 1; i <= 10; i++)
    {
        int w = r.Next() % 20+10;
        red = r.Next() % 256;
```



Фиг. 4.

```

        green = r.Next() % 256;
        blue = r.Next() % 256;
        Pen p = new Pen(Color.FromArgb(red,green,blue),w);
        g.DrawLine(p, 100, h, 600, h);
        h = h + w;
    }
}

```

Напишете отново всички програми от урока, като замените извикванията на DrawLine, при които като параметри се задават четирите цели координати, с извиквания при които параметри са началната и крайната точка на изчертаваната отсечка.

58

Упражнение

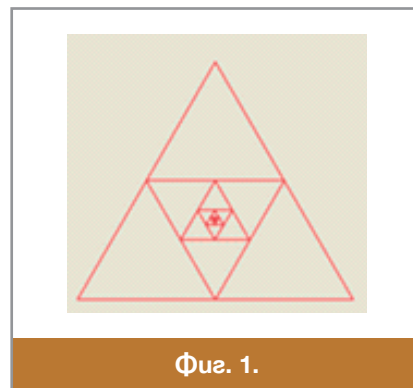
Задача 1: Да се изчертаят 6 вписани един в друг равностранни триъгълници (Фиг. 1).

Решение:

```

protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen pen = new Pen(Color.Red, 2);
    //изчертаване на първия триъгълник
    int x1 = 100, y1 = 500, side = 500;
    int x2 = x1 + side, y2 = y1;
    int h = (int)(side * Math.Sqrt(3) / 2);
    int x3 = (x1 + x2) / 2, y3 = y1 - h;
    g.DrawLine(pen, x1, y1, x2, y2);
    g.DrawLine(pen, x2, y2, x3, y3);
    g.DrawLine(pen, x3, y3, x1, y1);
    //изчертаване на вписаните триъгълници
    for (int i = 1; i <= 6; i++)
    {
        int x = x1, y = y1;
        x1 = (x1 + x2) / 2; y1 = (y1 + y2) / 2;
        x2 = (x2 + x3) / 2; y2 = (y2 + y3) / 2;
        x3 = (x3 + x) / 2; y3 = (y3 + y) / 2;
        g.DrawLine(pen, x1, y1, x2, y2);
        g.DrawLine(pen, x2, y2, x3, y3);
        g.DrawLine(pen, x3, y3, x1, y1);
    }
}

```



Фиг. 1.

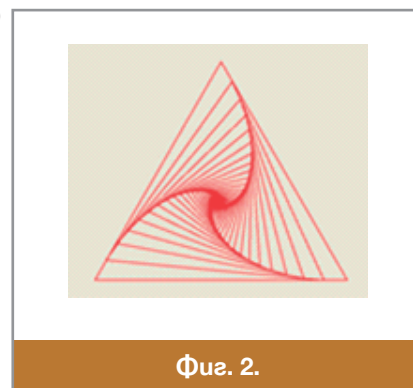
Задача 2. Да се изчертае картинката от Фиг. 2:

Решение:

```

protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen p = new Pen(Color.Blue, 2);
    //задаване на пропорции
    int n = 1, m = 30;
    //изчертаване на първия триъгълник
    int x1 = 100, y1 = 500, side = 500;
    int x2 = x1 + side, y2 = y1;
    int h = (int)(side * Math.Sqrt(3) / 2);
    int x3 = (x1 + x2) / 2, y3 = y1 - h;
    g.DrawLine(pen, x1, y1, x2, y2);
}

```



Фиг. 2.

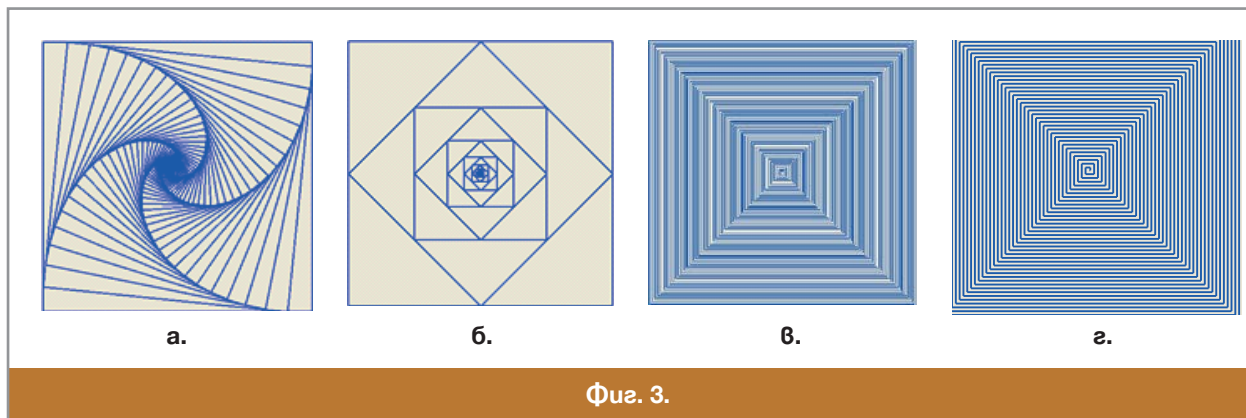

```

g.DrawLine(pen, x2, y2, x3, y3);
g.DrawLine(pen, x3, y3, x1, y1);
//изчертаване на вписаните триъгълници
for (int i = 1; i <= 80; i++)
{
    int x = x1, y = y1;
    x1 = (n*x1 + m*x2) / (n + m); y1 = (n*y1 + m*y2) / (n + m);
    x2 = (n*x2 + m*x3) / (n + m); y2 = (n*y2 + m*y3) / (n + m);
    x3 = (n*x3 + m*x) / (n + m); y3 = (n*y3 + m*y) / (n + m);
    g.DrawLine(pen, x1, y1, x2, y2);
    g.DrawLine(pen, x2, y2, x3, y3);
    g.DrawLine(pen, x3, y3, x1, y1);
}
}

```

Въпроси и задачи

1. Да се изчертае картинката от *Фиг. 3а*. **Упътване:** Направете промени в решението на задача 3.
2. Да се изчертае картинката от *Фиг. 3б*.
3. Да се изчертае квадратна развиваща се спираला (*Фиг. 3в* и *3г*) със страни, успоредни на координатните оси и начална точка в средата на екрана;
 - Първата страна е с дължина 1 и е ориентирана по положителната посока на абсисната ос;
 - Всяка следваща страна е с дължина с две по-голяма от предишната и е перпендикулярна на нея.
 - Посоката на развиване на спиралата е обратна на часовниковата стрелка.
 - Изчертаването продължава, докато се достигне до точка, която е извън чертожното поле.



Правоъгълник със страни успоредни на координатните оси

Правоъгълник може да се изчертае с четири извиквания на метода `DrawLine` (направете го за упражнение), но тъй като е най-разпространеният обект, то има специален метод за чертане на правоъгълници:

```

void DrawRectangle(Pen p, int x, int y,
                  int width, int height);

```

Писалката както при чертане на линии задава цвета и дебелината на линията на контура. Параметрите x и y съдържат координатите на горния ляв ъгъл на правоъгълника, а параметрите `width` и `height` – задават ширината (дължината на страната успоредна на абсцисата) и височината (дължината на страната успоредна на ординатата) на правоъгълника. Ясно е, че ако зададем ширина, която е равна на височината, методът ще изрисува квадрат. Ширината и височината се измерват в пиксели и трябва да са положителни числа (Фиг. 1а).

Изчертаване на елипса

За изчертаване на елипса се използва методът `DrawEllipse` със същите параметри, както при изчертаване на правоъгълник с метода `DrawRectangle`.

```
void DrawEllipse(Pen p, int x, int y,
                int width, int height);
```

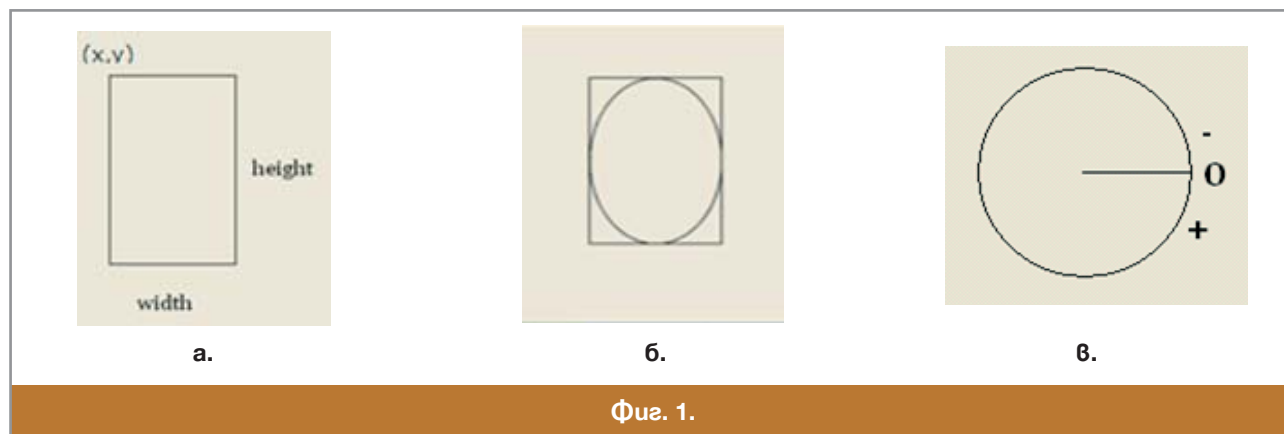
Това е така, защото се задава правоъгълникът, в който е вписана елипсата (Фиг. 1б). Ако правоъгълникът е квадрат, тогава методът `DrawEllipse` ще изчертае окръжност.

Изчертаване на дъга от елипса

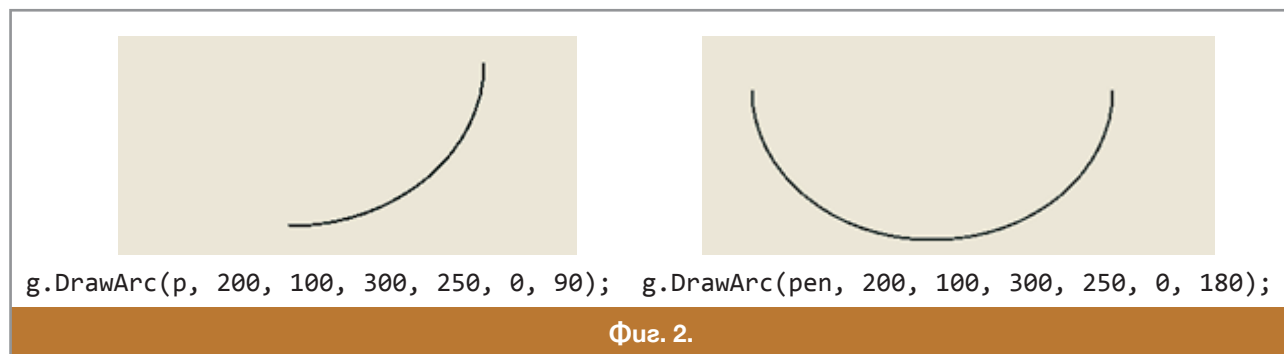
За да се изчертае дъга от елипса, трябва да се зададат същите параметри, които са необходими и за изчертаване на елипса, но допълнително трябва да се посочи къде започва и къде завършва дъгата. По тази причина методът за изчертаване на дъга от елипса `DrawArc` има следният вид:

```
void DrawArc(Pen p, int x, int y,
             int width, int height,
             int startAngle, int endAngle);
```

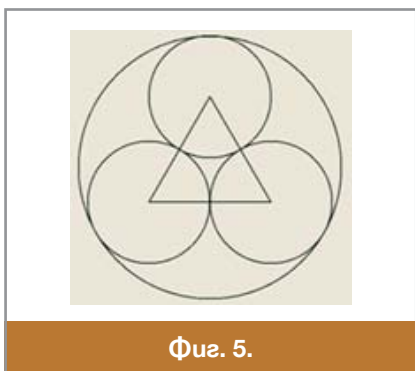
Първите пет параметъра са същите като на метода `DrawEllipse`. Двата допълнителни параметъра са ъглите, които задават началото и края на дъгата. Ъглите, които могат да бъдат положителни или отрицателни, се измерват в градуси **по посока на часовниковата стрелка**, с начало на хоризонталната ос надясно от центъра на елипсата (Фиг. 1в). На Фиг. 2 са показани две дъги, заедно с началния и крайния ъгъл на изчертаването.



Фиг. 1.



Фиг. 2.

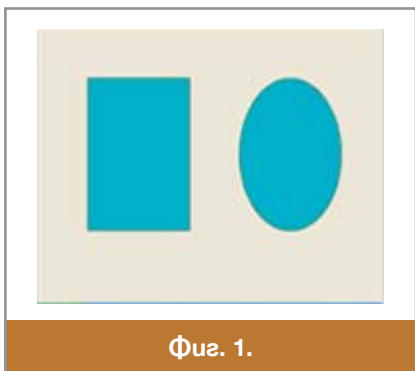


Напишете програма, която да изчертава чертежа показан на Фиг. 3 – равностранен триъгълник, разположен в центъра на чертожното поле, три взаимнодопирателни окръжности с центрове в трите върха на триъгълника и окръжността, до която е вътрешно се допират тези три окръжности (Фиг. 5).

60

Запълване на затворени фигури

Методи за запълване



Някои от методите на **Graphics**, разгледани дотук, дефинираха затворени области, но чертаеха само контура на областта и не запълваха вътрешността ѝ. Освен тези методи – с префикс на името **Draw**, които изчертават затворени области – съществуват и методи, имената на които започват с **Fill** (запълвам), които запълват вътрешността на съответната фигура. Първият параметър на тези методи е *четката* – обект от класа **Brush**, с която се запълва затворена от контур на фигура област. Ето как се създава нова четка:

```
Brush b = new SolidBrush(<цвет на четката>);
```

Препоръчително е, извикването на метод за запълване да се поставя преди извикването на метода за изчертаване контурите на съответната затворена фигура. Параметрите на метода с префикс на името **Fill** трябва да се съштите като параметрите на съответния му метод **Draw**.

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen p = new Pen(Color.Black, 2);
    Brush b = new SolidBrush(Color.Aqua);
    g.FillRectangle(b, 100, 100, 200, 300);
    g.DrawRectangle(p, 100, 100, 200, 300);
    g.FillEllipse(b, 400, 100, 200, 300);
    g.DrawEllipse(p, 400, 100, 200, 300);
}
```

Изчертаване и запълване на полигон

Полигоните са затворени фигури с три или повече страни, като триъгълници, четириъгълници, петъгълници и др., които не са непременно изпъкнали. За чертаене на полигони в **C#** се използва методът **DrawPolygon** на класа **Graphics**. Списъкът от параметри на този метод е по-различен от списъците на останалите разгледани до сега методи:

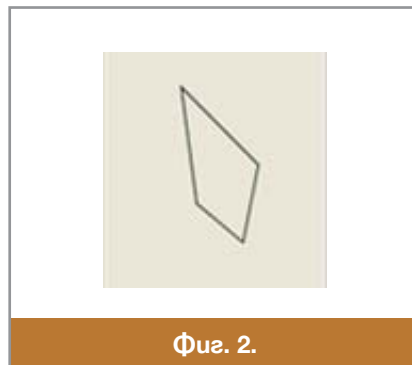
```
void DrawPolygon(Pen p, Point[] point);
```

Новото тук е масивът **point** от *точки* – елементи на класа **Point**. В този масив се задават коор-

динатите на върховете на многоъгълника в реда на обхождането им. Фигурата се затваря автоматично от отсечка, която свързва последната точка с първата. Например, да разгледаме следния масив от елементи на класа `Point`:

```
Point[] point = { new Point(100, 100), new Point(200, 200),
                 new Point(180, 300), new Point(120, 250) };
```

Той съдържа описанието на четири точки от чертожното поле. Ако извикаме метода `DrawPolygon` с този масив: `g.DrawPolygon(pen, point)`, той ще изчертае четириъгълника от *Фиг. 2*.



Работа с компютър

Задача: Да се изчертае звездата (петолъчка) от *Фиг. 3*.

Решение: Тъй като намирането на координатите на петте върха на звездата е трудна математическа задача, няма да се занимаваме с нея тук. Съответните формули са програмирани в цикъла `for`, с който се създава масива `point` с 5 елемента от класа `Point`. След края на цикъла се извика методът `DrawPolygon`, който изрисува фигурата:

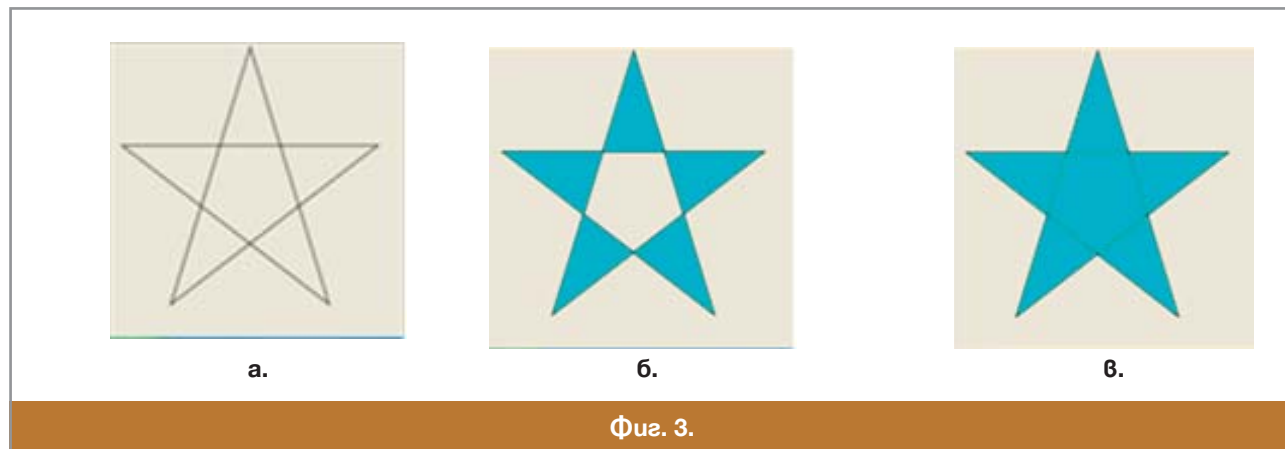
```
protected override void OnPaint(PaintEventArgs e)
{ Graphics g = e.Graphics;
  Pen p = new Pen(Color.Black, 2);
  int cx = ClientSize.Width;
  int cy = ClientSize.Height;
  Point[] point = new Point[5];
  for (int i = 0; i < 5; i++)
  { double angle = (i * 0.8 - 0.5) * Math.PI;
    point[i]=new Point((int)(cx*(0.25+0.24*Math.Cos(angle))),
                      (int)(cy*(0.5+0.48*Math.Sin(angle))));
  }
  g.DrawPolygon(p, point);
}
```

Ако се налага да се запълни получената фигура, трябва да се дефинира съответната четка и да се извика съответния метод за запълване, преди метода за изчертаване:

```
Brush b = new SolidBrush(Color.Aqua);
g.FillPolygon(b, point);
g.DrawPolygon(p, point);
```

Ако многоъгълникът не е изпъкнал, както е петолъчката, резултатът от запълването е неочакван (виж *Фиг. 3б*). Ако искате цялостно запълване (*Фиг. 3в*), трябва да се добави трети аргумент на метода за запълване:

```
g.FillPolygon(b, point, System.Drawing.Drawing2D.FillMode.Winding);
```





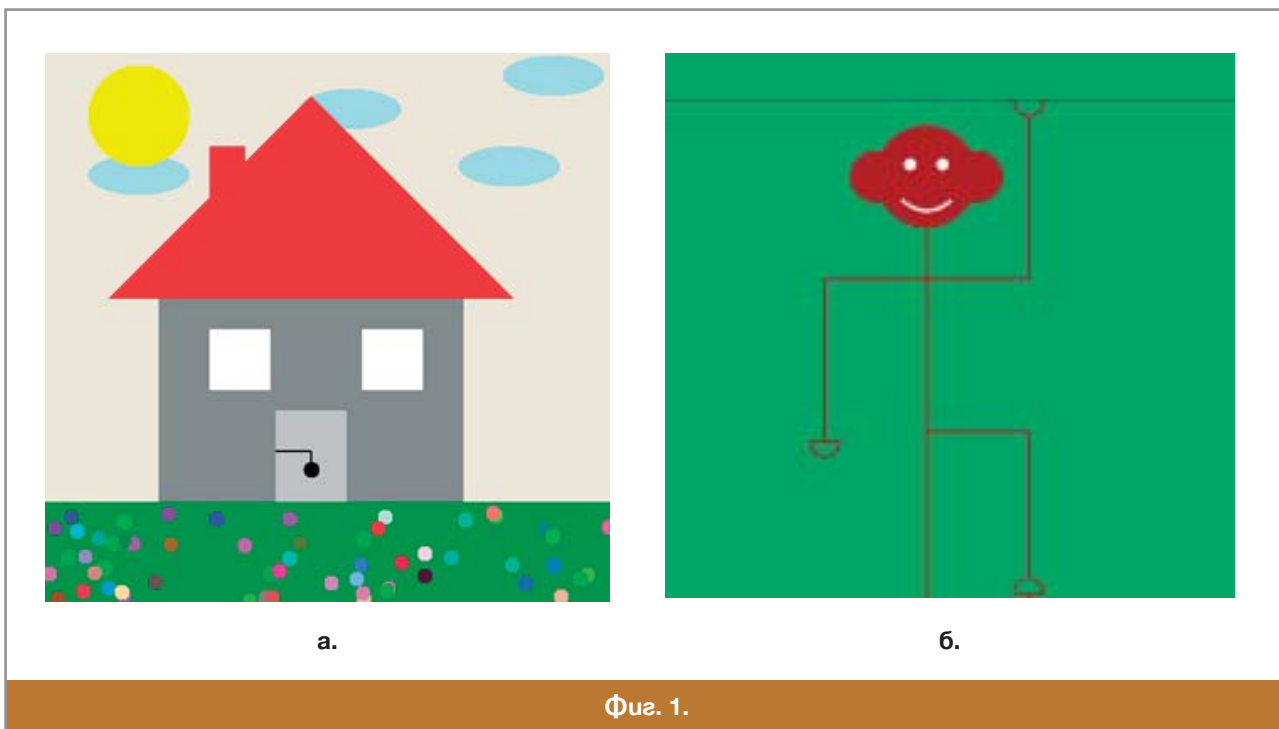
Напишете програма, която да изрисува няколко концентрични окръжности, с център средата на чертожното поле, със случайно генерирани радиуси и цветове (Фиг. 5).

61

Проект с компютърна графика

Задача 1. Напишете програма, която да изрисува картинката с къщата от Фиг. 1а.

Задача 2. Напишете програма, която да изрисува маймуната от Фиг. 1б.



62

Графика на математическа функция

От математика познаваме графиките на две функции – линейната, която представлява права линия и квадратната, която представлява парабола. Понякога, обаче, се налага да работим с функции, които не познаваме. В такъв случай бихме могли да получим по-ясна представа за тях, като разгледаме техните графики. Например, как би изглеждала графиката на функцията $y = x^3 - 5x^2 + 2x - 3$? Дали пресича абсцисната ос, т.е. има ли уравнението $x^3 - 5x^2 + 2x - 3 = 0$ реални корени? На тези и други подобни въпроси може да се даде отговор, ако визуализираме графиката на функцията.

За да се начертае графиката на функция, трябва да се зададат достатъчно много стойности за променливата x , в интересувания ни интервал (обикновено избран така, че да е симетричен относно нулата). За всяка такава стойност на x се изчислява съответната стойност на y . Така се получава последователност от точки в чертожното поле (независимо дали е на хартия или на екрана на компютъра), които са близо една до друга. Тези точки се свързват с отсечки и това визуализира графиката на функцията.

Сигурно е изненадващо, че графиката, съставена от отсечки от права линия, ще изглежда като крива линия. Но кривите линии, изчертавани от използваните вече методи за изчертаване на фигури, също са съставени от отсечки, които са много къси и така създават усещане за „изкривяване“ (подобно на фигурките, които сме рисували като малки, свързвайки последователно дадени точки).

За да се получи колкото може по-добро изображение, трябва да се работи с най-малката единица в дискретната компютърна координатна система – пикселът. Освен това, налага се да направим и съответната смяна на координатната система. Защото, както вече споменахме, математическата (Декартовата) координатна система и компютърната координатна система се различават по посоката на ординатната ос (виж *Фиг. 1*).

Да разгледаме точка P , с координати (dx, dy) в Декартова координатна система $(O_1X_1Y_1)$ и координати (x, y) в компютърната координатна система (OXY) (*Фиг. 2*). За да трансформираме Декартовата координатна система в компютърната, трябва да „изпратим“ нейното начало O_1 в центъра на чертожното поле, с координати

`(ClientSize.Width/2, ClientSize.Height/2)`.

Тогава координати на точката P в компютърната координатна система ще бъдат:

`x = ClientSize.Width/2 + dx` и `y = ClientSize.Height/2 + dy`.

Тези формули се наричат *формули за смяна на координатната система* и са много важни при изчертаването на графиката на функция!

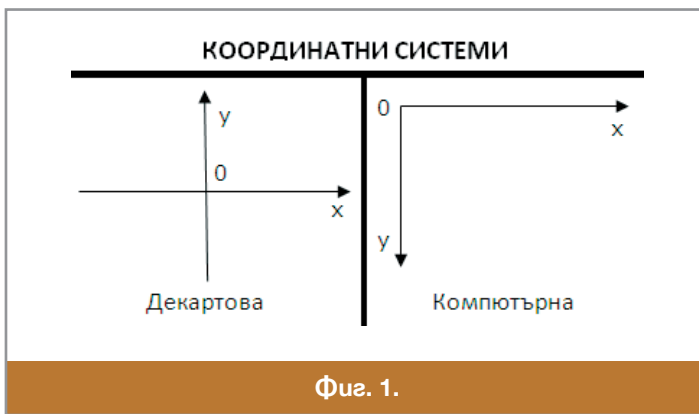
Остана да се извърши и мащабиране. Нека за мащаб да изберем големината в пиксели, с която искаме да се изчертае на екрана единичната отсечка от Декартовата координатна система. Ще съхраняваме мащаби в целочислената променлива m . Тогава формулите за смяна на координатната система стават:

`x = ClientSize.Width/2 + m*dx`, `y = ClientSize.Height/2 + m*dy`.

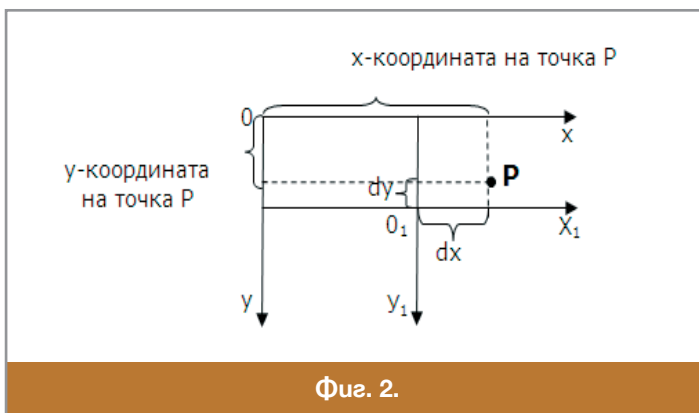
Сега, за да изчертаем графиката на функцията, ще даваме последователно стойности за аргумента x , започвайки от левия край на чертожното поле ($x = 0$) и ще увеличаваме стойността на x с по един пиксел, докато се достигне десния край на чертожното поле (`ClientSize.Width - 1`). За всяка такава стойност ще изчисляваме съответната стойност за y и ще изчертаваме отсечка от предишната точка до новата точка.

Ето как ще изглежда програмният код, чрез който ще се изчертае графиката на функцията $y = x^3 - 5x^2 + 2x - 3$:

```
protected override void
OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
```



Фиг. 1.

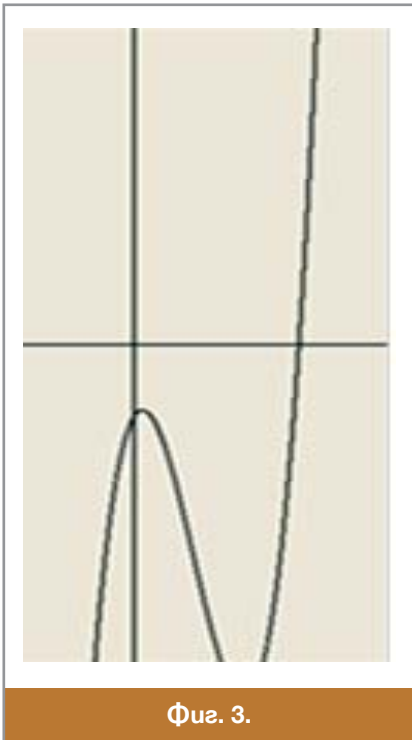


Фиг. 2.

```

Pen p = new Pen(Color.Black,2);
int x, y, ox = ClientSize.Width/2,
    oy = ClientSize.Height/2;
//изчертаване на координатните оси
g.DrawLine(p, 0, oy, ClientSize.Width, oy);
g.DrawLine(p, ox, 0, ox, ClientSize.Height);
double dx, dy, m = 20;
//избиране на началната точка (най-лявата)
dx=-ox/m;
dy = dx * dx * dx - 5 * dx * dx + 2 * dx - 3;
int xx=(int)(ox+m*dx), yy=(int)(oy-m*dy);
//последователно изчертаване на отсечки от графиката
for (x=1; x < ClientSize.Width; x++)
{
    dx = dx+1.0/m;
    dy = dx * dx * dx - 5 * dx * dx + 2 * dx - 3;
    x = (int)(ox + m * dx);
    y = (int)(oy - m * dy);
    g.DrawLine(pen, xx, yy, x, y);
    xx = x;
    yy = y;
}
}

```



Фиг. 3.

Визуализацията на графиката на функцията при избрания мащаб ($m = 20$) е показана на Фиг. 3. Програмата е доста сложна, затова ще дадем малко обяснения. Тъй като най-ляво разположените точки на компютърната координатна система са с координати $x = 0$, замествайки в първата формула за смяна на координатната система, получаваме $0 = ox + m*dx$ и $dx = -ox/m$. Тогава изчисляваме dy и избираме началната точка с координати $xx = ox + m*dx$ и $yy = oy + m*dy$.

Променливите x , y , xx и yy трябва да са целочислени, тъй като мярката, с която работим в екранната система е пиксел. Променливите dx и dy , разбира се, трябва да са дробни.

В цикъла, променливата dx се увеличава със стъпка $1.0/m$, което трансформирано в екранната система означава 1 пиксел. По този начин, на всяка стъпка на цикъла екранната абсциса се променя със стъпка от един пиксел. В променливите xx и yy се съхраняват координатите на последната изобразена точка, която е необходима за изчертаването на поредната отсечка.

Извеждане на текст

В някои изображения, например при изчертаване на графиката на функция, може да се добави и текст, например за означаване на осите, за заглавия и др. За извеждане на текст като графичен елемент се използва методът `DrawString`:

```
void DrawString(string s, Font f, Brush b, int x, int y);
```

Първият параметър в този метод е низ и съдържа текста, който ще се извежда. Вторият параметър е видът на шрифта, който ще се използва за изписване на текста – обект от класа `Font`. Засега ще използваме шрифта на формата, т.е. шрифта по подразбиране (`this.Font`).

Третият параметър показва цвета на буквите в шрифта. Най-простият начин да зададем цвят на текста е, да използваме 140-те цвята, дефинирани като свойства на класовете `Brushes` и `Color`. Последните два параметъра задават координатите на горния ляв ъгъл на полето, в което ще се изпише текстът. Например,

```
g.DrawString("Hello", this.Font, Brushes.Red, 500, 100);
```

ще изведе текста „Hello“ от позиция (500,100), с шрифта на формата, в червен цвят.

Ако искаме надписът да е с различен шрифт – по вид, цвят и размер, трябва предварително да създадем нов обект от класа Font:

```
Font font = new Font("Times New Roman", 24);  
g.DrawString("Hello", font, Brushes.Red, 500, 100);
```

Въпроси и загадки

1. Начертайте графиките на функциите:

а) $y = x^2 - 3x + 2$;

б) $y = \sqrt{x * x + 2 * x + 3}$;

в) $y = \sin(x)$.

63
64

Първи опити за анимация

Благодарение на това, че човешкото око може да различи около 24 кадъра за една секунда и на това, че компютърът може да извърши милионни операции за една секунда, може да се създава компютърна анимация, използвайки следния цикличен алгоритъм за анимиране на обекти:

1. Избира се място за изчертаване на обекта;
2. Избира се цвят на обекта и се изчертава;
3. Изчертава се обектът на същото място с цвета на фона;
4. Изчаква се определено време (в милисекунди);
5. Ако не е достигнато някакво условие за край на анимацията, то се връщаме на стъпка 1, иначе стъпка 6;
6. Край.

Работа с компютър

Задача 1: Да се визуализира преместване по екрана на оцветен правоъгълник.

Решение:

```
protected override void OnPaint(PaintEventArgs e)  
{ Graphics g = e.Graphics;  
  g.Clear(Color.White);  
  for (int x = 0; x < ClientSize.Width-200; x += 20)  
  { g.FillRectangle(Brushes.Blue, x, 200, 200, 300);  
    g.FillRectangle(Brushes.White, x, 200, 200, 300);  
    System.Threading.Thread.Sleep(50);  
  }  
}
```

Операторът `System.Threading.Thread.Sleep(<време в милисекунди>)`, както вече споменахме в предишен урок ще спре изпълнението на програмата за зададеното като параметър време, за да се даде време на човешкото око да възприеме един кадър, преди на екрана да бъде показан следващият.

Задача 2: Напишете програма, която да визуализира движението по билиардна маса на топка изпратена към някой от бордовете под ъгъл 45° или 135°

Решение: Много е важно да се отбележи, че когато при движението си билиардната топка се удари в ръба на масата под ъгъл 45° или 135°, тя отскача под ъгъл под ъгъл 135° или 45° съответно. Затова няма да има сложно пресмятане на координати, а само смяна на посоката на движение (умножаване с -1 на стъпката dx или dy, с която имитирате движението).

```
protected override void OnPaint(PaintEventArgs e)
```



```

{ Graphics g = e.Graphics;
  Pen p = new Pen(Color.Black,10);
  int cx = ClientSize.Width, cy = ClientSize.Height;
  g.FillRectangle(Brushes.Green, 0, 0,cx - 1, cy - 1);
  g.DrawRectangle(pen, 0, 0, cx - 1, cy - 1);
  Random r=new Random();
  int x=r.Next() % cx, y=r.Next() % cy;
  int dx=1, dy=1;
  while (1==1)
  { g.FillEllipse(Brushes.Black,x-10,y-10,20,20);
    System.Threading.Thread.Sleep(1);
    g.FillEllipse(Brushes.Green,x-10,y-10,20,20);
    if (x <= 15 || x >= cx - 16) dx = -dx;
    if (y <= 15 || y >= cy - 16) dy = -dy;
    x = x + dx; y = y + dy;
  }
}

```

Задача 3. Съставете и реализирайте проект за кратка анимация по ваш избор.

Речник

arc	ъък	арка; дъга от окръжност или елипса
fill	фил	запълвам
pen	пен	писалка
sleep	слийп	спя